

(19)



Europäisches Patentamt
European Patent Office
Office européen des brevets



(11) Publication number:

0 415 346 A2

(12)

EUROPEAN PATENT APPLICATION

(21) Application number: **90116473.1**

(51) Int. Cl.⁵: **G06F 9/44**

(22) Date of filing: **28.08.90**

(30) Priority: **29.08.89 US 400531**

(43) Date of publication of application:
06.03.91 Bulletin 91/10

(84) Designated Contracting States:
BE DE FR GB IT NL SE

(71) Applicant: **MICROSOFT CORPORATION**
One Microsoft Way
Redmond, Washington 98052-6399(US)

(72) Inventor: **Willman, Bryan M.**
14545 N. E. 43rd Place, No. 1308N
Bellevue, Washington, 98007(US)
Inventor: **Zbikowski, Mark J.**
15817 N. E. 178th Place
Woodinville, Washington 98072(US)
Inventor: **Letwin, James G.**
11428 Northeast 104th Street
Kirkland, Washington 98033(US)
Inventor: **Shah, Rajen Jayantilal**
517 123rd Avenue N.E.
Bellevue, Washington 98005(US)

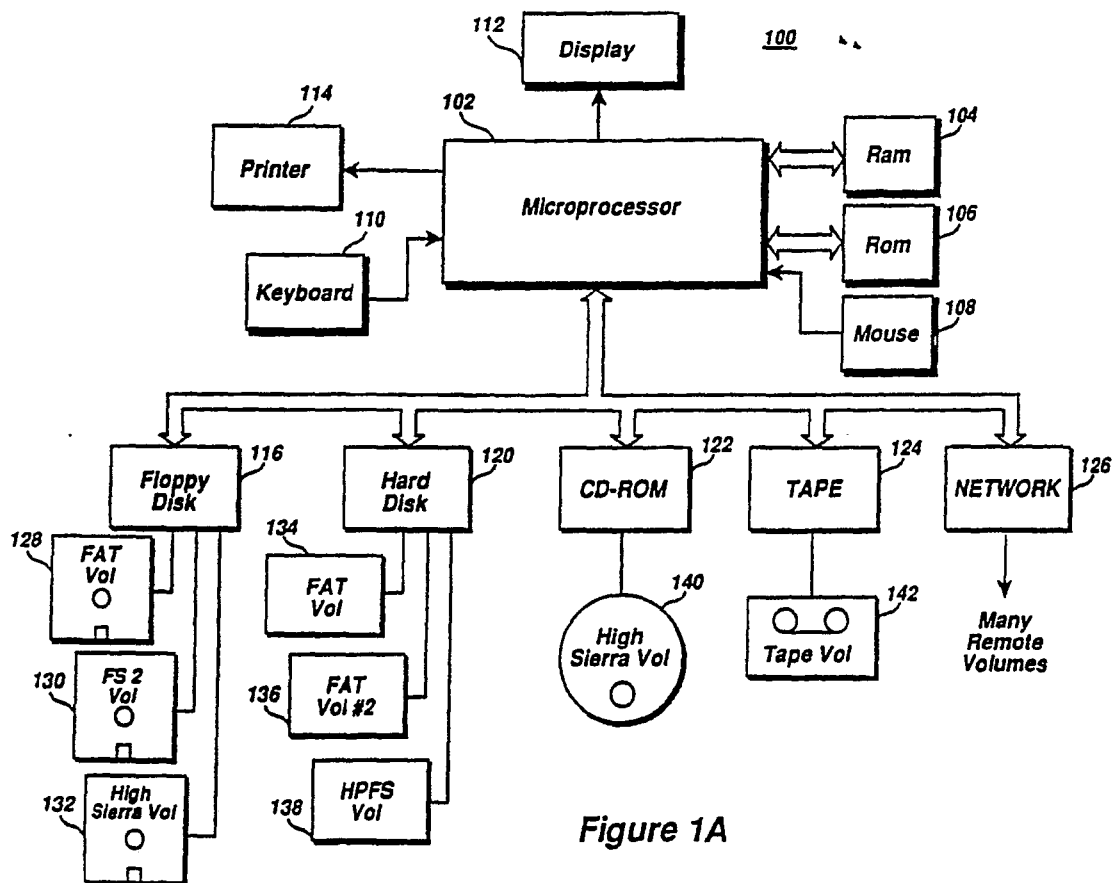
(74) Representative: **Patentanwälte Grünecker,**
Kinkeldey, Stockmair & Partner
Maximilianstrasse 58
W-8000 München 22(DE)

(54) **Method and system for dynamic volume tracking in an installable file system.**

(57) The invention comprises a method for mounting a file system for use in communicating with a data storage device and a computer system comprising the steps of:

- a) including a plurality of file system modules in a computer operating system including a default file system wherein said file systems are organized in a linked sequence;
- b) coupling a data storage device to said computer system;
- c) detecting a change in media in said data storage device or the first time said computer system accesses said data storage device;
- d) loading a file system identified in said list of file systems;
- e) reading a volume identifier from said media wherein the location of said volume identifier is specified by said loaded file system;
- f) comparing the read volume identifier from said media with the identifier associated with said file system;
- g) mounting said file system if said identifiers match;
- h) loading the next file system identified in said list of file systems if said identifiers do not match;
- i) returning to step (e) until each file system in said list of file systems has been tested or until a match is found; and
- j) mounting a default file system if no match is found.

EP 0 415 346 A2



METHOD AND SYSTEM FOR DYNAMIC VOLUME TRACKING IN AN INSTALLABLE FILE SYSTEM

Included in the specification is Appendix I, which is four sheets of microfiche containing 385 frames.

Field of the Invention

5

This invention relates to the field of computer control systems and more specifically to a method and means for facilitating communication between the devices which comprise a computer system.

10 Background of the Invention

Computer systems typically comprise a central processing unit, random access memory, read only memory, and a variety of peripheral devices such as data input devices, data output devices, and a variety of non-volatile data storage devices such as floppy disks and fixed or hard disks. Communication between
15 the respective devices in a computer system is typically controlled by a computer operating system. One well known computer operating system is the MS-DOS operating system available from Microsoft.

In the MS-DOS operating system, a single file system describes and defines the organization of files stored on peripheral devices. In order for the computer system to read or write data in a format recognized by both the computer system and the respective peripheral devices, data must be organized in accordance
20 with this file system. For example, in a conventional floppy disk peripheral device used with the MS-DOS operating system, data on a floppy disk is structured in accordance with a file system known as the FAT file system which is so named because of its use of file allocation tables. The FAT file system is one of the most widely used file systems in the world today. Other file systems may be associated with other types of data storage types of peripheral devices such as tape storage devices.

25 File systems facilitate communication between the operating system kernel and device dependant drivers and are responsible for converting read and write commands generated by an operating system kernel (as well as functions such as opening and closing files) into a form which may be recognized by the device driver.

When using the MS-DOS operating system, the operating system must be configured to define the
30 relevant file systems to be used with specific peripheral devices employed by the computer system. Once the file systems are defined, file systems remain static or unchanged unless the operating system is modified. This typically requires extensive programming effort and is typically quite time-consuming. It further requires extensive knowledge of the computer operating system and individuals who do not have access to operating system details can not easily modify the file systems.

35 Furthermore, in prior systems, disk media which contains files of foreign file systems may not be used with the native system. For example, over the years, many computer systems have been developed by a variety of manufacturers, each of which are based on alternate file system structures. With current static file system architectures, disk media from one system typically will not function with another type of system. As computers become more popular, it is increasingly important that files may be shared among all types of
40 computer systems. No system is known which allows disk media from virtually all known computer systems to be automatically recognized and read in a single operating environment. Further, no system is known which allows file systems to be added to a system or modified without the need for altering the computer operating system kernel.

45

Summary of the Invention

Briefly described, the present invention contemplates a method and means for automatically identifying media used with the computer system and for automatically and dynamically mounting a file system which
50 recognizes the media. In accordance with a preferred embodiment of the present invention, one or more data storage devices and a plurality of file system drivers are provided in a computer system including a default file system wherein the file systems are organized in a linked sequence. The computer system continuously monitors all peripheral devices in the system to detect any change in media in the peripheral storage devices. Whenever media in a data storage device is changed, or the first time the computer system accesses a data storage device, the first file system driver identified in the list of file system drivers

is loaded and a volume identifier is read from the media wherein the location of the volume identifier is specified by the loaded file system driver. The volume identifier read from the media is then compared with the identifier associated with the file system driver and the file system driver is mounted if the identifiers match. Otherwise, the next file system driver identified in the linked list of file system drivers is loaded.

5 The process is then repeated until each file system driver in the linked list of file system drivers has been tested or until a match is found. A default file system is mounted if no match is found.

Accordingly, it is an object of the present invention to provide a method and means for allowing a computer system to identify any of number of types of media and to mount the proper file system to be used with that media.

10 It is another object of the present invention for automatically mapping file systems to uncertain media.

It is yet another object of the present invention to provide a computer system which can automatically adapt to uncertain media without interaction from a user.

It is still another object of the present invention to provide an improvement to computer operating systems wherein file systems may be modified or added to the system without requiring modification of the operating system kernel.

15 It is another object of the present invention to provide a method and means for automatically mapping of file systems to uncertain media wherein all dependencies on the format of the media are encapsulated in the appropriate file system.

It is still another object of the present invention to provide a method and means for allowing a computer system to be booted from an arbitrary installable file system.

Brief Description of the Drawings

25 These and other objects may be fully appreciated through the detailed description of the invention below and the accompanying drawings in which:

Figure 1A is a block diagram of a computer system constructed in accordance with the principles of the present invention.

Figure 1B is a diagram showing the operating and file system architecture the system of Figure 1A.

30 Figures 2A is a diagram detailing the file system structure of the MS-DOS operating system.

Figure 2B is a diagram detailing the file system structure of the installable file system of the present invention.

Figure 3 is a more detailed diagram of the system of Figure 2B.

Figure 4 is a diagram showing the disk format of the FAT file system.

35 Figures 5A-5H are diagrams showing the disk format of an installable file system adapted for use with the present invention.

Figure 6 is a flow diagram detailing the overall operation of the mount process of the present invention.

Figure 7 is a diagram of the structure of the installable file system of the present invention.

40 Figure 8 is a flow diagram detailing the execution of name-based operations in accordance with the principles of the present invention.

Figure 9 is a flow diagram of the parsing process invoked by the named-based operations process.

Figure 10 is a flow diagram of the execution of handle-based operations in accordance with the principles of the present invention.

45 Figure 11 is a flow diagram of the FSH__DoVollo process invoked by the processes described in conjunction with Figures 8 and 10.

Detailed Description of the Invention

50 Figure 1 shows a computer system 100 which is constructed in accordance with the principles of the present invention. The system 100 comprises a central processing unit or microprocessor 102, random access memory 104, read only memory 106, input devices such as a mouse 108 and keyboard 110, output devices such as display 112 and printer 114 and a variety of non-volatile storage devices such as floppy disk drive 116, hard disk drive 120, CD-ROM drive 122, and tape drive 124. In addition, the computer system 100 is adapted for communicating with a network 126. Non-volatile storage means that data is present when the device is powered-off.

In prior systems, an operating system is statically configured with file system drivers wherein each peripheral device is compatible with only one media type and file system driver. If media is placed in a

drive which is not compatible with the designated file system driver, the media cannot be successfully accessed. The present invention provides a method and means for automatically mapping media to the file systems associated therewith independent of the peripheral device and without imposing any requirements on the format or location of data on the media, as will be further discussed below. For example, it is contemplated that the floppy drive unit 116 may be used with volumes formatted in accordance with a number of file systems wherein volume 128 is formatted in accordance with the FAT file system, volume 132 is formatted in accordance with the well known High Sierra file system and volume 130 is formatted in accordance with yet another file system. Similarly, various partitions of hard disk 120 may also be formatted in accordance with a number of file systems as indicated by volumes 134, 136 and 138. Similarly, the CD-ROM and tape system 124 may be used with volumes 140, 142, respectively, formatted with their own file systems. Further, network 126 may be coupled to any number of networks having servers which may operate in accordance with their own file systems.

The operation of the system 100 is coordinated by an operating system which may be any of a number of well known operating systems. However, the present invention is particularly adapted for use with the OS/2 operating system developed by Microsoft. The structure of the operating environment of the present invention is shown in Figure 1B. Typically, an application 152 generates file system requests which are processed by kernel 154. The kernel then routes this request to an appropriate file system driver (FSD) 156-170. Any file system driver may cooperate with a number of hardware devices. For example, the High Sierra file system 156 may be used with CD-ROM player 122 and disk drive 116 when performing file system operations on volumes 172, 174, respectively. Similarly, the FAT file system and the HPFS file systems may both be used for performing file system operations on volumes 176, 178, each of which are resident on hard disk 120. Further, the FAT file system driver may be used with disk drive 116 when performing file system operations on volume 180. Accordingly, the present invention provides a method and means for automatically and dynamically mapping uncertain media to the appropriate file system, regardless of the type and format of the file system.

Figure 2A shows the file system structure of the prior art MS-DOS operating system. In the MS-DOS operating system 200, the FAT file system 202 is embedded in the operating system kernel 204. Since the FAT file system is integrated into the system kernel, it is difficult to modify. Furthermore, if additional file systems are required, the operating system kernel 204 must be rewritten to accommodate them.

The present invention overcomes the above mentioned problems with the system architecture shown in Figure 2B. In the system 100, the OS/2 kernel 252 also includes the FAT file system 202 embedded therein. However, the present invention provides a method and means for dynamically attaching file system drivers 254, 256, 258 which are external to the operating system kernel 252. While the system 250 is shown with three installable file system drivers, the present invention is adapted to include a virtually unlimited number of file system drivers.

An installable file system driver (FSD) is analogous in many ways to a device driver. An FSD resides on the disk in a file that is structured like a dynamic-link library (DLL), typically with a SYS or IFS extension, and is loaded during system initialization by IFS = statements in the CONFIG.SYS file. IFS = directives are processed in the order they are encountered and are also sensitive to the order of DEVICE = statements for device drivers. This allows a user to load a device driver for a nonstandard device, load a file system driver from a volume on that device, and so on. Once an FSD is installed and initialized, the kernel communicates with it in terms of logical requests for file opens, reads, writes, seeks, closes, and so on. The FSD translates these requests using control structures and tables found on the volume itself into requests for sector reads and writes for which it can call special kernel entry points called File System Helpers (FsHlps). The kernel passes the demands for sector I/O to the appropriate device driver and returns the results to the FSD.

The procedure used by the operating system to associate volumes with FSDs is referred to as dynamic volume mounting and operates as follows. Whenever a volume is first accessed, or after it has been locked for direct access and then unlocked (for example, by a FORMAT operation), the operating system kernel presents identifying information from the volume to each of the FSDs in seriatim until an FSD recognizes the information. When an FSD claims the volume, the volume is mounted and all subsequent file I/O requests for the volume are routed to the FSD which claimed the volume.

This arrangement provides several advantages over the prior art. For example, if uncertain media is presented to the computer system, the computer system may scan the available file system drivers to locate a file system driver which recognizes the media thus providing for automatic mapping of file system driver to media. Furthermore, file system drivers may be updated without requiring a modification of the operating system kernel. In addition, as new types of peripheral devices are developed, appropriate file system drivers may be added to the operating system without disturbing existing system software.

A more detailed diagram of the system 100 is shown in Figure 3. The system 100 includes an operating

system kernel 252 which facilitates communication between an application program 302 and data storage devices such as disk device 304. The system 100 includes a device driver 306 which works in conjunction with a file system driver 254-258. While the system 100 is shown as including a single peripheral device 304, the present invention is adapted for use with any number of logical or physical peripheral devices.

5 In operation, the application program 302 issues logical file requests to the operating system kernel 252 by calling the entry points for the desired function. These functions may include requests to open files (DosOpen), to read files (DosRead), to write files (DosWrite), etc. The operating system kernel 252 passes these requests to the appropriate file system driver 254-258 for the particular volume holding the file. The appropriate installable file system driver then translates the logical file request into requests for reads or
10 writes of logical sectors of the designated media and calls an operating system kernel file system helper 308 to pass these requests to the appropriate device driver 306. File system helpers are discussed in more detail below. The disk driver 306 transforms the logical sector requests from the operating system kernel into requests for specific physical units: cylinders, heads and sectors of the media, and issues commands to the disk device to transfer data between disk media and random access memory 310.

15 The mapping of physical devices into particular file systems is discussed in further detail below. In the MS-DOS environment, floppy disks are referred to as volumes. Fixed disks (or hard disks) may be partitioned into multiple volumes. This terminology applies to the present invention as well. Briefly, whenever the system 100 is first booted, whenever a volume is first accessed, or whenever the system determines uncertain media is present in disk device 304, the system examines the first file system driver
20 in a linked list of file system drivers. If the file system driver recognizes the volume loaded in the disk device, the file system driver is mounted. Otherwise, the system sequentially polls the available file system drivers until a file system driver which recognizes the media is located. If no installable file system driver is found which recognizes the media of interest, a default file system driver is mounted. In the preferred practice of the present invention, the default file system is the FAT file system mentioned above.

25 Uncertain media may be detected in several ways. Many disk devices are provided with a mechanical latch mechanism which is exercised when a disk is ejected or installed in the disk device. The latch mechanism typically functions such that the next operation on the drive will indicate that the door has been opened. When the device driver receives this indication, ERROR_UNCERTAIN_MEDIA is returned to the operating system. In systems without mechanical latch mechanisms, it is assumed that media cannot be
30 changed in less than a predetermined time interval. In the preferred practice of the present invention, this interval is assumed to be two seconds. Thus if a particular volume has not been accessed for more than the predetermined interval, the media is presumed to be uncertain.

Figure 4 is a diagram of the disk format of the FAT file system. The FAT file system has been used with the MS-DOS operating system since its inception. A detailed description of the FAT file system may
35 be found in Duncan, "Advance MS DOS Programming", Microsoft Press, 1986, 1988. A brief description of the FAT file system follows. The FAT file system revolves around the File Allocation Table. Each logical volume is associated with its own FAT, which serves two important functions: it contains the allocation information for each file on the volume in the form of linked lists of allocation units and it indicates which allocation units are free for assignment to a file that is being created or extended.

40 When a disk is formatted in accordance with the FAT file system, a boot sector is written in sector zero. This is followed by one or more file allocation tables. The file allocation tables are followed by a root directory. The root directory is followed by the volume files. The boot sector contains various descriptive information about the volume in an area referred to as the boot parameter block or BPB, information such as a drive number and a volume I.D. as well as a bootstrap routine.

45 The file allocation table is divided into fields that correspond directly to the assignable clusters on a disk (clusters are power-of-2 multiples of sectors). These fields are typically 16 bits wide. The first two fields in the FAT are reserved. The first reserved FAT entry contains a copy of a media descriptor byte which is also found in the BPB. The remaining reserved fields contain OFFH. The remaining FAT entries describe the use of their corresponding disk clusters. Each file's entry in a directory contains the number of
50 the first cluster assigned to that file, which is used as an entry point into the FAT. From the entry point on, each FAT slot contains the number of the next cluster in the file, until a last-cluster mark is encountered. The FAT file system also provides for the option of maintaining a duplicate of the first file allocation table which may be used if access to a sector in the FAT fails due to a read error, etc.

Following the file allocation tables, is the root directory. The root directory contains 32 byte entries that
55 describe files, other directories, and an optional volume label.

The remainder of the volume after the root directory is known as the files area which may be viewed as pools of clusters, each containing one or more logical sectors. Each cluster has a corresponding entry in the FAT that describes its current use: available, reserved, assigned to a file, or unusable.

The FAT file system provides excellent performance with volumes which are less than 1 Mb. However, as volumes increase in size over 1 Mb, the performance of the FAT file system quickly degrades. This has become an increasingly severe problem as the size of readily available hard disks is rapidly increasing.

When volumes are less than 1 Mb, the FAT is small enough to be retained in random access memory at all times, thus allowing very fast random access to any part of a file. When applied to hard disks or fixed disks, however, the FAT becomes too large to hold in memory and must be paged into memory in pieces. This results in many superfluous disk head movements, thus degrading system throughput. In addition, since information about disk free space is dispersed across many sectors of FAT, it is impractical to allocate file space contiguously, and files become fragmented, further degrading system throughput. Furthermore, the use of relatively large clusters on hard disks results in much wasted space.

Figures 5A-5H are a series of diagrams showing the disk format of one instance of an installable file system. This file system is referred to as the high performance file system (HPFS). The high performance file system of the present invention eliminates the above-mentioned problems with the FAT file system and provides superior performance with all types of disk media. Referring now to Figure 5A, HPFS volumes can exist on a fixed disk along side of previously defined FAT partition types. HPFS volumes use a sector size of 512 bytes and have a maximum size of 2199 Gb (2^{32} sectors). While primarily designed for use with fixed disks, HPFS is compatible with virtually any type of disk media.

An HPFS volume is required to have very few fixed structures. Sectors 0-15 of a volume (8Kb) are allocated to the BootBlock 502 and contain a volume name field 504, a 32-bit volume ID field, a BIOS parameter block 508, and a disk bootstrap program 510. The disk bootstrap program 510 can be used in a restricted mode to locate and read operating system files wherever they may be found.

The BootBlock 502 is followed by a SuperBlock 512 and a SpareBlock 514. The SuperBlock 514 is only modified by disk maintenance utilities. It contains pointers 516 which point to free space bitmaps, a bad block list 518, a pointer 520 which points to a directory block band, and a pointer 522 which points to the root directory. It also contains a date field 524 which includes the date the volume was last checked and repaired with CHKDSK. CHKDSK is a well known OS/2 disk utility for detecting and cataloging bad portions of a disk.

The SpareBlock 514 contains various flags and pointers which will be further discussed below. It is modified as the system executes.

The remainder of the volume is divided into 8Mb bands, e.g. bands 516-522 which are used for storing files. While Figure 5A shows four 8 Mb bands, HPFS provides for a very large number of bands. Each band is provided with its own free space bitmap, see e.g. bitmaps 524-534. Each bit in the freespace bitmaps represents a sector. A bit is 0 if the sector is in use and 1 if the sector is available. The bitmaps are located at the head or tail of a band so that two bitmaps are adjacent between alternate bands. This allows the maximum contiguous free space that can be allocated to a file to be 16 Mb although the bitmap bandsize may be modified to accommodate files of virtually any size. One band, located at or towards the seek center of the disk, is called the directory block band and receives special treatment as will be further discussed below.

Every file or directory on an HPFS volume is anchored on a fundamental file system object called an Fnode which is shown in Figures 5B-5C. The Fnode 530 is the first sector allocated to a file or directory, and is pointed to by field 522 in the Superblock 504. Each Fnode occupies a single sector and contains control and access information field 540 used internally by the file system, an area 542 for storing extended attributes (EA) and access control lists (ACLs), a field 544 for indicating the length and the first 15 characters of the name of the associated file or directory, and an allocation structure 546 as shown in Figure 5B. An Fnode is always stored near the file or directory that it represents.

The allocation structure 546 shown in Figure 5C takes several forms, depending on the size and degree of continuity of the file directory. The HPFS of the present invention views a file as a collection of one or more runs or extents of one or more contiguous sectors. Each run is symbolized by a pair of double-words: a 32-bit starting sector number and a 32-bit length in sectors (this is referred to as run length encoding). From an application programs point of view, the extents are invisible; the file appears as a seamless stream of bytes.

The space reserved for allocation information in an Fnode can hold pointers to as many as eight runs of sectors of up to 16 Mb each. Reasonably small files of highly contiguous size can, therefore, be completely described within the Fnode.

The HPFS employs a new method to represent the location of files that are too large or too fragmented for the Fnode and consist of more than eight runs. The Fnode's allocation becomes the root for a B+ tree of allocation sectors, which in turn contain the actual pointers to the file's sector runs as shown in Figure 5D. The concept of B+ trees and B- trees is discussed in detail below. The Fnode's root has room for 12

elements. Each allocation sector can contain, in addition to various control information, as many as 40 pointers to sector runs. Therefore, a two level allocation B+ Tree can describe a file of 480(12*40) sector runs, with a theoretical maximum size of 7.68 Gb (12*40*16 Mb) in the preferred practice of the present invention.

5 In the unlikely event that a two-level B+ Tree is not sufficient to describe a highly fragmented file, the HPFS file system introduces additional levels in the tree as required. Allocation sectors in the intermediate levels can hold as many as 60 internal (nonterminal) B+ tree nodes, which means that the descriptive ability of this structure rapidly grows to numbers that are extremely large. For example, a three-level allocation B+ Tree can describe as many as 28,800 (12*60*40) sector runs.

10 Run-length encoding and B+ Trees of allocation sectors are a memory efficient way to specify a file's size and location and offer several significant advantages over the prior art. Translating a logical file offset into a sector number is extremely fast: the file system merely traverses the list (or B+ Tree of lists) of run pointers, summing up run sizes until the correct range is found. It can then identify the sector within the run with a simple calculation. Run-length encoding also makes it trivial to extend the file logically if the newly
15 assigned sector is contiguous with the file's previous last sector; the file system merely increments the size double-word of the file's last run pointer and clears the sector's bit in the appropriate freespace bitmap.

Directories, like files, are anchored on Fnodes. A pointer 522 to the Fnode for the root directory is found in the SuperBlock 512. Figure 5E shows the directory structure of the present invention wherein a directory Fnode 550 is shown. The Fnodes for directories other than the root are reached through subdirectory
20 entries in their parent directories.

Directories are built up from 2 Kb directory blocks, which are allocated as four consecutive sectors on the disk and can grow to any size. See e.g. directory blocks 552, 554, 556. The file system attempts to allocate directory blocks in the directory band, which is located at or near the seek center of the disk. Once the directory band is full, the directory blocks are allocated wherever space is available.

25 Each 2 Kb directory block may contain from one to many directory entries. See e.g. entries 558-568. A directory entry contains several fields, including a field 570 for time and date stamps, a field 572 which contains an Fnode pointer, a usage count field 574 for use by disk maintenance programs (which are well known), a field 576 which contains the length of the file or directory name, a field 578 for the name itself, and a field 580 which contains B- Tree pointer, as shown in Figure 5E. Each directory entry begins with a
30 word 582 that contains the length of the entry. This provides for a variable amount of flex space at the end of each entry, which can be used by special versions of the file system and allows the directory block to be traversed very quickly.

The number of entries in a directory block varies with the length of names. If the average filename length is 13 characters, an average directory block will hold approximately 40 entries. The entries in a
35 directory block are sorted by the binary lexical order of their name fields. The last entry is a dummy record that marks the end of the block.

When a directory gets too large to be stored in one block, it increases in size by the addition of 2 Kb blocks that are organized as a B- Tree. When searching for a specific name, the file system traverses a directory block until it either finds a match or finds a name that is lexically greater than the target. In the
40 latter case, the file system extracts the B- Tree pointer from the entry. If this pointer points to nowhere, the search failed; otherwise, the file system follows the pointer to the next pointer to the next directory block in the tree and continues the search.

Assuming 40 entries per block, a two-level tree of directory blocks can hold 1640 directory entries and a three level tree can hold 65,640 entries. In other words, a particular file can be found (or shown not to
45 exist) in a typical directory of 65,640 files with a maximum of three disk accesses. The actual number of disks accesses depends on cache contents and the location of the file's name in the directory block B-Tree. This presents a vast improvement over the FAT file system where in a worst case, 4,000 sectors would have to be read to establish whether a file was present in a directory containing the same number of files.

50 The B- Tree directory structure of the HPFS has interesting implications beyond its effect on open and find operations. A file creation, renaming, or deletion may result in a cascade of complex operations, as directory blocks are added or freed or names are moved from one block to the other to keep the tree balanced. In fact, a rename operation could fail for lack of disk space even though the file itself is not growing. In order to avoid this problem, the HPFS reserves a small pool of free blocks that can be drawn
55 from in a directory emergency; a pointer to this pool is preferably stored in the SpareBlock.

File attributes are information about a file that is maintained by the operating system outside the file's overt storage area.

The HPFS of the present invention supports Extended Attributes (EAs) taking the form

name = value

except that the value portion can be either a null-terminated (ASCII) string or binary data. In the preferred practice of the present invention, each file or directory can have a maximum of 64 Kb of EAs attached to it although this limit may be readily modified.

- 5 The storage method for EAs can vary. If the EAs associated with a given file or directory are small enough, they will be stored in the Fnode. If the total size of the EAs is too large, they are stored outside the Fnode in sector runs, and a B+ Tree of allocation sectors is created to describe the runs. If a single EA gets too large, it may be pushed outside the Fnode into a B+ Tree of its own.

- 10 The present invention provides an improvement to the OS/2 kernel API functions DosQFileInfo and DosSetFileInfo that allow application programs to manipulate extended attributes for files. The present invention further provides two new functions DOSQPathInfo and DosSetPathInfo which may be used to read or write the EAs associated with arbitrary pathnames. An application program may either request the value of a specific EA (supplying a name to be matched) or can obtain all of the EAs for the file or directory at once. The support of EAs facilitates the use of object oriented application programming. Information of
15 almost any type can be stored in EAs, ranging from the name of the application that owns the file, names of dependent files, icons, and executable code.

- The HPFS attacks potential bottlenecks in disk throughput at multiple levels. It uses advanced data structures, contiguous sector allocation, intelligent caching, read-ahead, and deferred writes in order to boost performance. First, the HPFS matches its data structures to the task at hand: sophisticated data
20 structures (B- Trees and B+ Trees) for fast random access to filenames, directory names, and lists of sectors allocated to files or directories, and simple compact data structures (bitmaps) for locating chunks of free space of the appropriate size. The routines that manipulate these data structures are preferably written in assembly language.

- The main objective of the HPFS is to assign consecutive sectors to files whenever possible. The time
25 required to move the disk's read/write head from one track to another far outweighs the other possible delays, so the HPFS avoids or minimizes such head movements by allocating file space contiguously and by keeping control structures such as Fnodes and freespace bitmaps near the things they control. Highly contiguous files also help the file system make fewer requests of the disk driver for more sectors at a time, allow the disk driver to exploit the multisector transfer capabilities of the disk controller, and reduce the
30 number of disk completion interrupts that must be serviced.

- Keeping files from becoming fragmented in a multitasking operating system in which many files are being updated concurrently is a feature not found in the prior art. One strategy the HPFS uses is to scatter newly created files across the disk in separate bands, if possible, so that the sectors allocated to the files as they are extended will not be interleaved. Another strategy is to preallocate 4Kb of contiguous space to the
35 file each time it must be extended and give return any excess when the file is closed.

If an application knows the ultimate size of a new file in advance, it may assist the HPFS by specifying an initial file allocation when it creates a file. The system then searches all the free space bitmaps to find a run of consecutive sectors large enough to hold the file. That failing, it searches for two rounds that are half the size of the file, and so on.

- 40 The HPFS relies on several different kinds of caching to minimize the number of physical disk transfers it requests. It caches sectors, as did the FAT file system. But unlike the FAT file system, the HPFS manages very large caches efficiently and adjusts sector caching on a per-handle basis to the manner in which a file is used. The HPFS also caches pathnames and directories, transforming disk directory entries in to an even more compact and efficient in memory representation.

- 45 Another technique that the HPFS uses to improve performance is to preread data it believes the program is likely to need. For example, when a file is opened, the file system will preread and cache the Fnode and the first few sectors of the file's contents. If the file is an executable program or the history information in the file's Fnode shows that an open operation has typically been followed by an immediate sequential read of the entire file, the file system will preread and cache much more of the file's contents.
50 When a program issues relatively small read requests, the file system always fetches data from the file in 2Kb chunks and caches the excess, allowing most read operations to be satisfied from the cache.

- The HPFS of the present invention relies heavily on lazy writes based on OS/2 multitasking capabilities (sometimes called deferred writes or write behind) to improve performance. For example, when a program requests a disk write, the data is placed in the cache and the cache buffer is flagged as dirty (that is,
55 inconsistent with the state of the data on disk). When the disk becomes idle or the cache becomes saturated with dirty buffers, the file system uses a captive thread from a daemon process to write the buffers to disk, starting with the oldest data. Captive threads and daemon processes are described in a series of texts: Hastings, et al. "Microsoft OS/2 Programmers Reference", Microsoft Press, 1989.

In general, lazy writes mean that programs run faster because their read requests will typically not be stalled waiting for a write request to complete. For programs that repeatedly read, modify, and write a small working set of records, it also means that many unnecessary or redundant physical disk writes may be avoided. Lazy writes have their certain dangers, and therefore, the present invention provides that a
 5 program can defeat them on a per-handle basis by setting the write-through flag in the OpenMode parameter for DosOPen, or it can commit data to disk on a per-handle basis with the DosBufReset function. Both DosOpen and DosBufReset functions are available in current versions of OS/2.

The extensive use of lazy writes makes it imperative for the HPFS to be able to recover gracefully from write errors under any but the most dire circumstances. For example, by the time a write is known to have
 10 failed, the application has long since gone on its way under the illusion that it has safely shipped the data into disk storage. The errors may be detected by hardware (such as a "sector not found" error returned by the disk adapter), or they may be detected by the disk driver in spite of the hardware during a read-after-write verification of the data.

The primary mechanism for handling write errors is referred to as a hotfix. When an error is detected,
 15 the file system takes a free block out of a reserved hotfix pool, writes the data to that block, and updates the hotfix map. (The hotfix map is simply a series of pairs of doublewords, with each pair containing the number of a bad sector associated with the number of its hotfix replacement.) A copy of the hotfix map is then written to the SpareBlock, and a warning message is displayed to let the user know that there is a problem with the disk device.

Each time the file system requests a sector read or write from the disk driver, it scans the hotfix map and replaces any bad sector numbers with the corresponding good sector holding the actual data.

One of CHKDSK's duties is to empty the hotfix map. For each replacement block on the hotfix map, it allocates a new sector that is in a favorable location for the file that owns the data, moves the data from the hotfix block to the newly allocated sector, and updates the file's allocation information (which may involve
 25 rebalancing allocation trees and other elaborate operations). It then adds the bad sector to the bad block list, releases the replacement sector back to the hotfix pool, deletes the hotfix entry from the hotfix map, and writes the updated hotfix map to the SpareBlock.

The HPFS maintains a Dirty FS flag in the SpareBlock of each HPFS volume. The flag is cleared when all files on the volume have been closed and all dirty buffers in the cache have been written out or, in the
 30 case of the boot volume, when Shutdown has been selected and has completed its work.

During the OS/2 boot sequence, the file system inspects the DirtyFS flag on each HPFS volume and, if the flag is set, will not allow further access to that volume until CHKDSK has been run. If the DirtyFS flag is set on the boot volume, the system will run CHKDSK automatically.

In the event of a truly major catastrophe, such as loss of the SuperBlock or the root directory, the HPFS
 35 is designed to give data recovery the best possible chance of success. Nearly every type of crucial file object, including Fnodes, allocations sectors, and directory blocks, is doubly linked to both its parent and its children and contains a unique 32-bit signature. Fnodes also contain the initial portion of the name of their file or directory. Consequently, SHODS can rebuild an entire volume by methodically scanning the disk for Fnodes, allocations sectors, and directory blocks, using them to reconstruct the files and directories and
 40 finally regenerating the freespace bitmaps.

As mentioned above, the present invention employs B+ trees and B- trees (binary trees) for logically ordering files and directories. Binary trees are a technique for imposing a logical ordering on a collection of data items by means of pointers, without regard to the physical order of the data.

Referring now to Figures 5F, in a simple binary tree, each node contains some data, including a key
 45 value that determines the node's logical position in the tree, as well as pointers to the node's left and right subtrees. The node that begins the tree is known as the root; the nodes that sit at the ends of the tree's branches are sometime called the leaves.

To find a particular piece of data, the binary tree is traversed from the root. At each node, the desired key is compared with the node's key; if they don't match, one branch of the node's subtree or another is
 50 selected based on whether the desired key is less than or greater than the node's key. This process continues until a match is found or an empty subtree is encountered as shown in Figure 5F.

Such simple binary trees, although easy to understand and implement, have disadvantages in practice. If keys are not well distributed or are added to the tree in a non-random fashion, the tree can become quite asymmetric, leading to wide variations in tree traversal time.

In order to make access times uniform, many programmers prefer a particular type of balanced tree
 55 known as a B- Tree as shown in Figure 5. The important points about a B- Tree are that the data is stored in all nodes, more than one data item might be stored in a node, and all of the branches of the tree are of identical length.

The worst-case behavior of a B- Tree is predictable and much better than that of a simple binary tree, but the maintenance of a B- Tree is correspondingly more complex. Adding a new data item, changing a key value, or deleting a data item may result in the he splitting or merging of a node, which in turn forces a cascade of other operations on the tree to rebalance it.

5 As shown in Figure 5G, a B+ Tree is a specialized form of B- Tree that has two types of nodes: internal, which only point to other nodes, and external, which contain the actual data.

The advantage of a B+ Tree over a B- Tree is that the internal nodes of the B+ Tree can hold many more decision values than the intermediate-level nodes of a B- Tree, so the fan out of the tree is faster and the average length of a branch is shorter. This compensates for the fact that a B+ Tree branch must be
10 followed to its end to find the necessary data, whereas in a B- Tree the data may be discovered at an intermediate node or even at the root.

The present invention comprises an improvement to the OS/2 operating system and may be implemented with many of the utilities and subroutines available in current versions of OS/2. While primarily intended for use with the OS/2 operating system, the principles of the present invention may be applied to
15 virtually any computer operating system. With the exception of the new utilities and subroutines described herein, all other utilities and subroutines are currently available and well known. For a detailed description of the OS/2 operating system refer to the OS/2 Programmer's Reference texts described above. Volume Management in the improved OS/2 operating system of the present invention is responsible for the same duties it performed in previous versions of OS/2 such as detecting when the wrong volume is inserted in the
20 drive, detecting when a volume has been removed, generating new information on new media that has been placed in the drive via the Volume Parameter Block (VPB), communicating with the appropriate device drivers, providing the system with device information needed to access new inserted media, interfacing with the Buffer and CDS mechanisms, and informing the system of changes to a specific volume.

In previous versions of OS/2, there was only one file system. The present invention provides for
25 multiple file systems in a unified environment. The volume manager determines which file system should have access to a particular volume, provides mechanisms that will allow file system drivers (FSDs) to manage their resources for a particular volume, and provides the same support for all FSDs provided in the past for managing volumes. The present invention relies on existing well-known OS/2 calls as well as several new functions described herein. A complete description of the installable file system of the present
30 invention is set forth in Appendix I which is attached hereto in the form of microfiche and is herein incorporated by reference.

The present invention contemplates the use of MOUNT and UNMOUNT processes to facilitate the identification and loading of the correct file system driver for individual volumes.

The MOUNT Process gets initiated by several different events:

- 35 1. The first access to a volume.
2. Whenever the volume in a drive becomes uncertain. (This usually means the user put a new medium in the drive.)
- 3 Whenever access to a volume that is not in the drive is requested.

Input to the MOUNT process is a pointer to a drive parameter block (DPB) which is used to do I/O to
40 the device driver and to store the handle to the VPB for the volume currently believed to be in the drive. A mount operation updates this. A local VPB is allocated on a stack and initialized with the DPB pointer.

Referring now to Figure 6, the MOUNT process 600 begins by reading logical sector 0 of the media as indicated by item 602. Any errors encountered from the device driver are ignored because it is possible that different types of media (i.e Optical Disk or CD-ROM) may have track 0 unreadable. Before reading logical
45 sector 0 the temporary mount buffer is initialized to zeros. The Volume label text field is initialized to "UNLABELED". Sector 0 is checked to determine whether the format is recognized by comparing signature byte for a special value (41). If the format is not recognized, the information pertinent to the VPB is filled in on the stack (i.e 32 Bit Volume Serial Number).

A BUILDVPB call is then issued by item 604 to the device driver specified in the DPB. BUILDVPB is a
50 procedure exported by a device drivers. A detailed description of the BUILDVPB procedure is set forth in Appendix I. BUILDVPB is called to learn the physical parameters of the device (bytes per sector, sectors per track, and the like.) The device driver is passed a pointer to the buffer that contains information it can use to determine the physical parameters of the volume. For most drivers this is sector 0, for some very old ones it is the first sector of the FAT. If the device is not able to interpret the data read from Sector 0 (for
55 example, the floppy in question is not FAT, so the FAT ID byte is meaningless) the device returns a minimal BPB, adequate to allow the kernel and FSDs to do necessary I/O to completely identify the volume.

The relevant fields from the previously created BPB are copied into the Local VPB on the stack (i.e Sectors/track, NumberofHeads, Total Sectors, Sector Size). A new VPB is allocated and information from

the Local VPB is copied into it. The present invention then enters loop 606 to poll each FSD by calling the FS_MOUNT (flag=0) entry point with the handle of newly created VPB, a pointer to logical sector 0, and pointers to VPB file system independent and dependent areas of the VPB as indicated by item 608. The FSD may call FSH_DoVolIO to read other sectors from the volume (It must allocate its own buffer). If the FSD returns ERROR_UNCERTAIN_MEDIA, the error is returned and the process is restarted as indicated by decision 610. If the FSD supports boot sectors, it may check the file system name field in the boot sector to determine whether it recognizes it. If the FSD does not support boot sectors I/O to the device is performed to determine if the FSD recognizes the volume. Once an FSD has recognized the volume it updates the relevant fields in the VPB file system independent and dependent areas as indicated by item 612. The VPB file system independent and dependent areas are discussed in more detail in conjunction with Figure 7. At this time the FSD issues a FS Helper (FSH) function to determine whether the new volume is the same as any of the other volumes that the present invention manages. This FS Helper returns pointers to the file system independent and dependent areas. The FSD then copies information from the newly created VPB to old VPB as indicated by item 614. The newly created VPB is destroyed after the MOUNT call. The FSD then performs any cleanup work on the old VPB such as invalidating any buffers since the volume may have been removed from the drive.

Once an FSD has recognized the volume, the present invention eliminates the new VPB if a match is found in the list. Otherwise, the VPB is linked into a list of mounted FSDs. If no FSDs are recognized, the VPB is freed and the FAT file system is mounted as indicated by decision 614 and item 616.

When a new volume is inserted into a drive and the old volume has no more kernel references to the old volume the present invention issues a FS_MOUNT (flag=2) to the FSD so that resources allocated to that volume may be de-allocated.

When the present invention detects that a newly inserted volume is different than the last volume in the drive a FS_MOUNT (flag = 1) call is issued to the FSD so that any cleanup type work such as buffer invalidation on the removed volume may be performed. If there are no more kernel references to the volume, a FS_MOUNT (flag = 2, UNMOUNT) will follow. If the newly inserted volume is the same as the last seen volume in the drive, this call is not issued.

The present invention contemplates the use of an efficient mechanism to utilize existing kernel resources for functions required by an FSD. Specifically, if an FSD requires a function existing within the kernel, the FSD issues a file system helper (FSH) call which invokes the file system helper. The called FSH then returns the requested information. A brief summary of file system helpers is set forth below. While the summary set forth below lists several important file system helpers, it is contemplated that additional file system helpers will be provided as required. File system helpers are discussed in detail in Appendix I.

File System Helpers:

FSH_GETVOLPARM - On many FS calls, the handle to the VPB is passed to the FSD and it is often necessary for the FSD to access the file system independent and dependent areas of the VPB. This helper provides that service.

FSH_DOVOLIO - When an FSD needs to perform I/O to a specified volume it uses this helper to insure that the requested volume is indeed in the drive, to call the appropriate device driver and to handle hard errors. This helper may be used at all times within the FSD. When called within the scope of a FS_MOUNT call, it applies to the volume in the drive. However, since volume recognition is not complete until the FSD returns to the FS_MOUNT call, the FSD must take care when an ERROR_UNCERTAIN_MEDIA is returned. This indicates that the media has gone uncertain while trying to identify the media in the drive. This may indicate that the volume that the FSD was trying to recognize was removed. In this case, the FSD releases any resources attached to the hVPB passed in the FS_MOUNT call and ERROR_UNCERTAIN_MEDIA is returned to the FS_MOUNT call. This directs the volume tracking logic to restart the mount process.

FSH_DUPLICATEVPB - During a FS_MOUNT call the input VPB may be the same volume as one of the other volumes being managed. It is the responsibility of the FSD generate up-to-date information on the new volume and copy that information to the older duplicate VPB. This helper determines if an older duplicate VPB exists and if it does, pointers to the file system independent and dependent areas of the older duplicate VPB will be returned so that these areas can be updated by the FSD. The FSD then performs any cleanup work on the old volume since the volume may have been removed.

As mentioned above, the present invention contemplates the use of pre-existing OS/2 resources whenever possible. The listing below is a summary of the hierarchy of functions invoked during the

operation of the present invention.

5	1	DoVolIO
	1.1	WhatVolume
	1.1.1	ProbeChange
	1.1.2	ResetMedia
10	1.1.3	GenhVPB
	1.1.3.1	LockVBuf
	1.1.3.2	ReadBoot
15	1.1.3.3	BuildBPB
	1.1.3.4	FSMountVolume
	1.1.3.4.1	Bmp_Get
20	1.1.3.4.2	VPBCopy
	1.1.3.4.3	VPBLink
	1.1.3.4.4	VPBFind
	1.1.3.4.5	VPBFree
25	1.1.3.5	SetVPB
	1.1.3.6	FindVID
	1.1.3.7	DiskIO
30	1.1.3.8	CRC
35	1.1.3.9	VPBFIND
	1.1.3.10	Bmp_Get
	1.1.3.11	VPBCopy
	1.1.3.12	VPBLink
40	1.1.3.13	UnlockVBuf
	1.1.3.14	BufInvalidate (Redetermine Media)
	1.1.3.15	FlushBuf (Redetermine Media)
45	1.1.4	IncVPBRef
	1.1.5	DecVPBRef
	1.1.5.1	VPBFree
	1.1.6	ResetCurrency
50	1.1.6.1	NextCDS
	1.1.6.2	PointComp
	1.1.6.3	BufInvalidate

55 **Table 1.**

The present invention is invoked whenever media becomes uncertain or whenever media is first accessed. The volume management function of the present invention is represented by line 1. The initial process is to determine what volume has been presented to the system as indicated by line 1.1. In line

1.1.1, ProbeChange is called to access the device driver to determine if the device driver detected a change in media. If a change in media was detected, ResetMedia is invoked in line 1.1.2 to instruct the device driver to allow I/O to the media. GenhVPB is then invoked in line 1.1.3 to generate a volume parameter block. This process begins with line 1.1.3.1 where LockVBuf is called to clear and serialize a buffer in the operating system kernel. In line 1.1.3.2, the data in the media boot sector is read into the operating system buffer. The system proceeds to line 1.1.3.3 wherein BuildBPB is invoked to call the disk driver and build a boot parameter block. FS_Mount is then invoked in line 1.1.3.4. The first step in FS_Mount invokes Bmp_Get in line 1.1.3.4.1 which is a memory management utility in the kernel which is called to set-up a buffer for the BPB. In line 1.1.3.4, when FSMountVolume is called, it iterates through the list of FSDs, calling each FSD's FS_Mount procedure until one returns success or the end of the list is reached. If an FSD returns success, in line 1.1.3.4.2, VPBCopy is called to create a temporary buffer for a copy of the BPB. VPBLink is then called in line 1.1.3.4.3 to link the VPB into a chain and set-up the BPB to point to the next VPB in the chain and to initialize the current VPB to the beginning of the list. VPBFind is invoked in line 1.1.3.4.4 to examine the chain of VPBs to find a VPB which possesses the same volume identifier as the VPB being processed. If a duplicate VPB identifier is found, VPBfree is called in line 1.1.3.4.5 to free the VPB under examination from the BPB if a duplicate VPB is found in the list of VPBs. Once FSMountVolume is complete, SetVPB is invoked in line 1.1.3.5 which sets up the appropriate fields in the VPB. In line 1.1.3.6, FindVID is called to find the volume identifier. DiskIO is invoked in line 1.1.3.7 if no boot block is found in sector 0 of the media to locate the BPB for the volume. If no FSD's FS_Mount routine returned success, then inline code which is logically equivalent to the FS_Mount procedure for the (resident) FAT file system is called. In line 1.1.3.8 CRC is called to checksum the first directory of old FAT volumes, to generate a unique volume serial number for volume that do not have a serial number in their boot sectors. The functions listed in lines 1.1.3.9 -1.1.3.13 are then invoked to generate a new volume identifier and free the volume identifier buffer. In line 1.1.2.14, BuflInvalidate is invoked to invalidate all data in the buffer if the media has changed since the process began. If so, FlushBuf is called in line 1.1.3.15 to flush the buffers for the new media.

If a preexisting VPB for the volume was not found, IncVPBRef in line 1.1.4 is invoked to increment a reference counter for the current VPB which is used to record whether the volume of interest is still open to the operating system kernel. In line 1.1.5, DecVPBRef is invoked to decrement the reference counter for a previous VPB. If the reference counter is decremented to zero, VPBFree is invoked in line 1.1.5.1 to free the VPB. ResetCurrency is called in line 1.1.6 to mark position data in current directory structures as invalid. NextCDS (1.1.6.1) and PointComp (1.1.6.2) are internal routines used to enumerate current directory structures (CDSs). In line 1.1.6.3 BuflInvalidate is called to remove (now stale) VPB references from a file system buffer pool.

As mentioned above, a VPB is used by the system to store information about a particular volume that is in use in the system. A volume is defined as a medium in a block device and the information on the medium differentiates it from every other volume.

VPBs are kept in a segment as BMP. Therefore, the system needs only track the records that are in use, and takes manages the free list.

Every time a new volume is encountered, i.e a VPB built for a volume does not match any of the VPBs already in the system, a new entry is allocated in the BMP managed segment and is filled in with the relevant data from the medium. Every time the system is finished with a VPB, i.e. its refcount goes to zero, the entry in the BMP managed segment is freed, BMP tracks this freed storage for reuse. The structures used by the functions of Table I are set forth below.

A VPB is divided into three parts:

1. the kernel private part, used to keep information the kernel needs to manage the VPB (reference counts, for example). This is private to the kernel, meaning that FSDs never access or modify it.
2. the file system independent part, used by all file systems and independent of any particular file system. This is passed to an installable file system (IFS) for certain file system (FS) calls, and
3. a part that is specific to the file system that is using the VPB. This is set out as a "work area" that the file system can use as required. This is passed to the IFS for certain FS calls. The layout of the VPB is shown in Figure 7.

The following structure defines the file system independent part of the VPB. This structure is used by all file systems irrespective of the type of file system.

```

5      vpbfsi      STRUC
      vpi_ID       DD      ?      ; 32 bit unique ID of file
      vpi_pDPB     DD      ?      ; Drive volume is in
      vpi_cbSector DW      ?      ; Size of physical sector in bytes
      vpi_totsec   DD      ?      ; Total number of sectors on medium
      vpi_trksec   DW      ?      ; Sectors per track on medium
      vpi_nhead    DW      ?      ; Number of heads in device
      vpi_text     DB      VPBTEXTLEN DUP (?) ; printable ID for users
      vpbfsi      ENDS

```

10

The following structure defines the file system dependent part of the VPB. This structure is used by file systems as they see fit.

```

15      vpbfsd     STRUC
      vpd_work     DB      VPDWORKAREASIZE DUP (?)
      vpbfsd      ENDS

```

The following structure defines the structure of the volume parameter block (VPB).

20

```

      vpb          STRUC
      Fields used by kernel for all file systems
      vpb_flink    DW      ?      ; handle of forward link
      vpb_blink    DW      ?      ; handle of back link
25      vpb_IDsector DD      ?      ; sector number of ID

      vpb_ref_count DW      ?      ; count of objects that point to VPB
30      vpb_search_count DW      ?      ; count of searches that point to VPB
      vpb_first_access DB      ?      ; This is initialized to -1 to force a media
      vpb_signature DW      ?      ; Signature which specifies VPB validity
      vpb_flags     DB      ?      ; flags
      vpb_FSC       DD      ?      ; Pointer to the file system control block (FSC).

```

35

The following fields are used for file system dependent work.

```

40      vpb_fsd     DB      SIZE vpbfsd DUP (?)

```

The following fields are used for file system independent work.

```

45      vpb_fsi     DB      SIZE vpbfsi DUP (?)
      vpb_fsi      ENDS

```

The following structure is used by FSH_GETVOLPARM - which is used to get VPB data from VPB handle.

```

50      ENTRY      push      word hVPB          (1 word)
                  push      dword ptr to file system ind. (2 word)
                  push      dword ptr to file system dep. (2 word)
                  call      FSHGETVOLPARM

```

55

```

      EXIT      (ax) = return code
                  0 - success

```

EP 0 415 346 A2

The following structure is used by FSH_DOVOLIO - which is used for volume-based sector-oriented transfers

```

5      ENTRY          push      word Operation          (1 word)
                        push      word hVPB              (1 word)
                        push      dword ptr to user transfer area (2 word)
                        push      dword ptr to sector count   (2 word)
                        push      dword starting sector number (2 word)
                        call      FSHDOVOLIO
10
      EXIT  (ax) = return code
          0 - success

```

15 The following structure is used by FSH_DUPLICATEVPB - which is used to get VPB data to a duplicate (old) VPB.

```

20      ENTRY          push      word hVPB              (1 word)
                        push      dword ptr to file system ind. (2 word)
                        push      dword ptr to file system dep. (2 word)
                        call      FSHGETVOLPARM
25
      EXIT  (ax) = return code
          0 - success

```

30 RedetermineMedia has a special set of entry parameters, as shown, below.

```

      ENTRY (DS;SI) point to dpb
35      EXIT          Carry clear = >
                      (DS;SI).hVPB is filled in with the "correct" volume
                      Carry Set = >
                      (AX) = I/O packet status; operation was failed
      USES            AX, BX, DX, DI, ES, Flags
40

```

The following calls are used for volume management intra-component interfaces.

GenhVPB is used to determine the internal VPB in a particular drive. Any errors returned are sent to the user.

45 Inputs; ds;si point to DPB of interest. It and whatever volume was in it last are locked.

Outputs; Carry clear = > ax is handle to VPB for drive

Carry set = > operation generated an error

zero clear = > operation was failed

zero set = > nested uncertain media occurred

50 ;

All registers may be modified

BuildBPB is called to generate a valid BPB for an old disk; one that does not have a recognized boot sector. The newer disks have a KNOWN and VALID BPB in the boot sector. The buffer to the device driver is part of the BuildBPB call.

55 Inputs; ds;si point to DPB of interest

pVPBBuf is locked

Outputs; carry clear = >

ds;si points to a BPB

carry set = >
 (AX) = status word from device
 zero set = > nested uncertain media error
 zero reset => operation was failed
 5 All registers modified all except BP
 FSMountVolume checks to determine whether an IFS Driver recognizes the Volume of interest.
 FSMountVolume Loops through the FSD chain calling each FS Driver FS__Mount entry point to
 determine whether the IFS recognizes the volume of interest. The loop terminatea when the first IFS
 recognizes the volume or when the loop counter for the number of FS Drivers installed in the system
 10 decrements to 0.
 ;
 Inputs; ds;bx point to pVPBBuf boot sector
 di offset of LocalVPB on Stack
 Outputs; di = offset to FSC If an IFS recognized the volume.
 15 di = -1 if no IFS driver recognized the volume
 ax = vpb handle
 Registers modified: ax,bp,bx,di,es,si,ds
 VPBFree removes the VPB from the link list and Frees its block from the segment.

20
 ENTRY (BP) = handle to VPB
 EXIT VPB unlinked and Freed
 USES bx,bp,cx,di,ds,es

25
 VPBLink inserts the new VPB at the beginning of the list and adjusts the forward and backlink fields of
 new VPB and the old first VPB.

30
 ENTRY ES;DI = New VPB
 EXIT VPB Linked into list.
 USES DS,SI

35
 VPBFind scans the internal list looking for a VPB with the same Vol. ID as the input VPB.

40
 ENTRY DS;SI = Pointer to input VPB Vol. ID
 EXIT AX = hVPB if found
 AX = 0 if not found

 USES AX,BX,CX,DI,DS,ES

45
 VPBCopy copies a VPB from the local area to the BMP managed area and stamps VPB as valid.

50
 ENTRY SI = Offset of LocalVPB on Stack
 ES;DI -> New VPB
 EXIT None
 USES AX,CX,DS,SI

55
 Volume management, i.e., detecting when the wrong volume is mounted and notifying the operator to
 take corrective action, is handled directly through the operating system kernel and the appropriate device
 driver. According to the principles of the present invention, each file system driver (FSD) generates a
 volume label and 32-bit volume serial number for each volume used with the system. Preferably, these are
 stored in a reserved location in logical sector zero when a volume is formatted. No particular format is

required to store this information. The operating system kernel calls the FSD to perform operations that might involve it. The FSD updates the volume parameter block (VPB) whenever the volume label or serial number is changed.

When the FSD passes an I/O request to an FS helper routine the device driver passes the 32-bit volume serial number and the volume label (via the VPB). When the I/O is performed on a volume, The operating system kernel compares the requested volume serial number with the current volume serial number it maintains for the device. This is an in-storage test (no I/O required) performed by checking the Drive Parameter Block's (DPB) VPB of volume mounted in the drive. If unequal, The operating system kernel signals the critical error handler to prompt the user to insert the volume having the serial number and label specified.

When a media change is detected a drive, or the first time a drive is accessed on behalf of an application program interface (API) function call, the present invention determines the FSD (file system driver) that will be responsible for managing I/O to that volume. The present invention then allocates a VPB (volume parameter block) and polls the installed FSDs an FSD indicates that it does recognize the media. The FSDs are polled as described above.

The FAT FSD is the last in the list of FSDs and, by recognizing all media, will act as the default FSD when no other FSD recognition takes place.

According to the principles of the present invention, there are two classes of file system drivers:

1. an FSD which uses a block device driver to do I/O to a local or remote (virtual disk) device. (This is referred to as a local file system), and
2. an FSD which accesses a remote system without a block device driver This is called a remote file system.

The connection between a drive letter and a remote file system is achieved through a programmatic interface. The DosFsAttach system call is used to create a binding between an object in the system name space (e.g. A drive) and an FSD.

The connection between a pseudo-character device and a remote file system is also achieved through the DosFsAttach interface. The DosFsAttach interfaces comprises the DosFsAttach and DosQFsAttach calls which are described in detail in Appendix I.

When a local volume is first referenced, the present invention sequentially asks each local FSD in the FSD chain to accept the media, via a call to each FSD's FS__MOUNT entry point. If no FSD accepts the media then it is assigned to the default FAT file system. Any further attempt made to access an unrecognized media other than by FORMAT, results in an 'INVALID__MEDIA__FORMAT' error message.

Once a volume has been recognized, the relationship between drive, FSD, volume serial number, and volume label is stored. The volume serial number and label are stored in the volume parameter block, (VPB). The VPB is maintained by the operating system for open files (file-handle based I/O), searches, and buffer references.

Subsequent requests for a removed volume require polling the installed FSDs for volume recognition by calling FS__MOUNT. The volume serial number and volume label of the VPB returned by the recognizing FSD and the existing VPB are compared. If the test succeeds, the FSD is given access to the volume. If the test fails, the operating system signals the critical error handler to prompt the user for the correct volume.

The connection between media and VPB is saved until all open files on the volume are closed, search references and cache buffer references are removed. Only volume changes cause a re-determination of the media at the time of next access.

Access to an operating system partition on a bootable, logically partitioned media is through the full operating system function set such as the function set available with the OS/2 operating system. A detailed description of disk partitioning design is available in the OS/2 Programmer's Reference texts described above.

The present invention provides the DosQFsAttach function to identify remote devices which communicate with the operating system through a network.

The purpose of DosQFsAttach is to query information about an attached remote file system, a local file system, about a character device, or about pseudo-character device name attached to a local or remote FSD.

The sequence for calling DosQFsAttach is as follows:

EXTRN DosQFsAttach:FAR

```

5      PUSH ASCIIZ      DeviceName    ; Device name or 'd:'
      PUSH WORD         Ordinal       ; Ordinal of entry in name list
      PUSH WORD         FSAInfoLevel ; Type of attached FSD data required
      PUSH OTHER        DataBuffer    ; Returned data buffer
      PUSH WORD         DataBufferLen ; Buffer length
      PUSH DWORD        0             ; Reserved (must be zero)
      CALL DosQFsAttach

```

10

Where:

DeviceName points to the drive letter followed by a colon, or points to a character or pseudo-character device name, or is ignored for some values of FSAInfoLevel. If DeviceName is a drive, it is an ASCIIZ string having the form of drive letter followed by a colon. If DeviceName is a character or pseudo-character device name, its format is that of an ASCIIZ string in the format of a filename in a subdirectory called which is preferably designated \DEV\.

Ordinal is an index into the list of character or pseudo-character devices, or the set of drives. Ordinal always starts at 1. The Ordinal position of an item in a list has no significance. Ordinal is used strictly to step through the list. The mapping from Ordinal to item is volatile, and may change from one call to DosQFsAttach to the next.

FSAInfoLevel is the level of information required, and determines which item the data in DataBuffer refers to.

Level 0x0001 returns data for the specific drive or device name referred to by DeviceName. The Ordinal field is ignored.

Level 0x0002 returns data for the entry in the list of character or pseudo-character devices selected by Ordinal. The DeviceName field is ignored.

Level 0x0003 returns data for the entry in the list of drives selected by Ordinal. The DeviceName field is ignored.

DataBuffer is the return information buffer, it is in the following format:

30

```

      struct {
          unsigned short iType;
          unsigned short cbName;
          unsigned char  szName[];
          unsigned short cbFSDName;
          unsigned char  szFSDName[];
          unsigned short cbFSADData;
          unsigned char  rgFSADData[];
      };

```

35

40

iType type of item

```

1 = Resident character device
2 = Pseudo-character device
3 = Local drive
4 = Remote drive attached to FSD

```

45

```

cbName    Length of item name, not counting null.
szName    Item name, ASCIIZ string.
cbFSDName Length of FSD name, not counting null.
szFSDName Name of FSD item is attached to, ASCIIZ string.
cbFSADData Length of FSD Attach data returned by FSD.
rgFSADData FSD Attach data returned by FSD.

```

50

szFSDName is the FSD name exported by the FSD, which is not necessarily the same as the FSD name in the boot sector.

For local character devices (iType = 1), cbFSDName = 0 and szFSDName will contain only a terminating NULL byte, and cbFSADData = 0.

For local drives (iType = 3), szFSDName will contain the name of the FSD attached to the drive at the

time of the call. This information changes dynamically. If the drive is attached to the operating system kernel's resident file system, szFSDName will contain "FAT" or "UNKNOWN". Since the resident file system gets attached to any disk that other FSDs refuse to MOUNT, it is possible to have a disk that does not contain a recognizable file system, but yet gets attached to the resident file system. In this case, it is possible to detect the difference, and this information helps programs in not destroying data on a disk that was not properly recognized.

DataBufferLen is the byte length of the return buffer. Upon return, it is the length of the data returned in DataBuffer by the FSD.

10

Returns: IF ERROR (AX not = 0)

AX = Error Code:

15

ERROR_INVALID_DRIVE - the drive specified is invalid
 ERROR_BUFFER_OVERFLOW - the specified buffer is too short for the
 returned data.
 ERROR_NO_MORE_ITEMS - the Ordinal specified refers to an item not in
 the list.
 ERROR_INVALID_LEVEL - invalid info level

Information about all block devices and all character and pseudo-character devices is returned by DosQFsAttach. The information returned by this call is highly volatile.

Preferably, calling programs should be aware that the returned information may have already changed by the time it's returned to them. The information returned for disks that are attached to the kernel's resident file system can be used to determine if the kernel definitely recognized the disk as one with its file system on it, or if the kernel just attached its file system to it because no other FSDs mounted the disk.

The set of error codes for errors general to all FSDs is 0xEE00 - 0xEEFF. The following errors have been defined although others may be added as needed:

ERROR_VOLUME_NOT_MOUNTED = 0xEE00 - The FSD did not recognize the volume.

The set of error codes which are defined by each FSD are 0xEF00 - 0xFEFF.

Disk media and file system layout are described by the following structures. The data which are provided to the file system may depend on the level of file system support provided by the device driver attached to the block device. These structures are relevant only for local file systems.

/* file system independent - volume params */

```
struct vpsi {
    unsigned long vpi_vid;           /* 32 bit volume ID */
    unsigned long vpi_hDEV;          /* handle to device driver */
    unsigned short vpi_bsize;        /* sector size in bytes */
    unsigned long vpi_totsec;        /* total number of sectors */
```

45

```
    unsigned short vpi_trksec;       /* sectors / track */
    unsigned short vpi_nhead;        /* number of heads */
    char vpi_text[12];               /* asciiz volume name */
};
```

50

```
/* file system dependent - volume params */
struct vpsd {
    char vpd_work[36]; /* work area */
};
```

55

As mentioned above, the FS_MOUNT function is called to mount and unmount volumes and its purpose is to examine volumes to determine whether an FSD it recognizes the file system format. The sequence for calling FS-Mount is as follows:

```

5      int far pascal FS_MOUNT (flag, pvpfsi, pvpfsd, hVPB, pBoot)
      unsigned short flag;
      struct vpfsi far * pvpfsi;
      struct vpfsd far * pvpfsd;
      unsigned short hVPB;
10     char far * pBoot;

```

Where:

flag indicates operation requested.

15 flag = 0 indicates that the FSD is requested to mount or accept a volume.

flag = 1 indicates that the FSD is being advised that the specified volume has been removed.

flag = 2 indicates that the FSD is requested to release all internal storage assigned to that volume as it has been removed from its drive and the last kernel-managed reference to that volume has been removed.

20 flag = 3 indicates that the FSD is requested to accept the volume regardless of recognition in preparation for formatting for use with the FSD.

All other values are reserved. The value passed to the FSD will be valid.

pvpfsi - A pointer to file-system-independent portion of VPB. If the media contains an operating system-recognizable boot sector, then the vpi_vid field contains the 32-bit identifier for that volume. If the media does not contain such a boot sector, the FSD generates a unique label for the media and places it into the
25 vpi_vid field.

pvpfsd - pointer to file-system-dependent portion of VPB. The FSD may store information as necessary into this area.

hVPB - handle to volume.

30 pBoot - pointer to sector 0 read from the media. This pointer is ONLY valid when flag == 0. The buffer the pointer refers to MUST NOT BE MODIFIED. The pointer is always valid and does not need to be verified when flag == 0; if a read error occurred, the buffer will contain zeroes.

The FSD examines the volume presented and determines whether it recognizes the file system. If so, it returns zero after having filled in appropriate parts of vpfsi and vpfsd. The vpi_vid and vpi_text field are filled in by the FSD. If the FSD has an operating system format boot sector, it converts the label from the
35 media into asciiz form. The vpi_hDev field is filled in by the operating system. If the volume is unrecognized, the driver returns non-zero.

The vpi_text and vpi-vid are updated by the FSD each time these values change.

The contents of the vpfsd are as follows:

FLAG = 0

40 The FSD issues an FSD_FINDDUPHVPB to determine whether a duplicate VPB exists. If one exists the VPB fs dependent area of the new VPB is invalid and the new VPB is unmounted after the FSD returns from the FS_MOUNT call. The FSD updates the fs dependent area of the old duplicate VPB.

If no duplicate VPB exists the FSD initializes the fs dependent area.

FLAG = 1

45 VPB fs dependent part is same as when FSD last modified it.

FLAG = 2

VPB fs dependent part is same as when FSD last modified it.

After media the recognition process, the volume parameters may be examined using the FSH_GETVOLPARM call. The volume parameters should not be changed after the media recognition
50 process.

During a mount request, the FSD may examine other sectors on the media by using FSH_DOVOLIO to perform the I/O. If an uncertain-media return is detected, the FSD is "cleans up" and returns ERROR_UNCERTAIN_MEDIA to allow the volume mount logic to restart on the newly-inserted media. The FSD provides the buffer to use for additional I/O.

55 The operating system kernel manages the VPB via the refcount counter mentioned above. All volume-specific objects are labelled with the appropriate volume handle and represent references to the VPB. When all kernel references to a volume disappear, FS_MOUNT is called with flag = 2, indicating a dismount request.

When the kernel detects that a volume has been removed from its drive, but there are still outstanding references to the volume, FS_MOUNT is called with flag = 1 to allow the FSD to store clean (or other regenerable) data for the volume. Data which is dirty and cannot be regenerated is retained so that the data may be written to the volume when it is remounted in the drive. For the purposes of the present invention, clean data is data which is unchanged and dirty data is data which has been modified.

When a volume is to be formatted for use with an FSD, the operating system kernel calls the FSD's FS_MOUNT entry with flag = 3 to allow the FSD to prepare for a format operation. The FSD accepts the volume even if it is not a volume of the type that FSD recognizes, since format changes the file system on the volume. The operation may be failed if formatting cannot be completed. (For example, an FSD which supports only CD-ROM.)

Since most computer system hardware does not allow for kernel-mediated removal of media, it is certain that the unmount request is issued when a volume is not present in any drive.

FSH_DOVOLIO performs I/O to a specified volume. FSH_DOVOLIO formats a device driver request packet for the requested I/O, locks the data transfer region, calls the device driver, and reports any errors to the hard error daemon before returning to the FSD. Any retries indicated by the hard error daemon or actions indicated by DOSERROR are done within the call to FSH_DOVOLIO.

The following describes the calling format for FSH_DOVOLIO.

```

int far pascal FSH_DOVOLIO (operation, hVPB, pData, pcSec, iSec)
    unsigned short    operation;
    unsigned short    hVPB;
    char far *    pData;

```

```

    unsigned short far * pcSec;
    unsigned long    iSec;

```

Where:

The operation bit mask indicates read/read-bypass/write/write-bypass/verify-after-write/write-through and no-cache operation to be performed.

Bit 0x0001 off indicates read.

Bit 0x0001 on indicates write.

Bit 0x0002 off indicates no bypass.

Bit 0x0002 on indicates cache bypass.

Bit 0x0004 off indicates no verify-after-write operation.

Bit 0x0004 on indicates verify-after-write.

Bit 0x0008 off indicates errors signalled to the hard error daemon.

Bit 0x0008 on indicates hard errors will be returned directly.

Bit 0x0010 off indicates I/O is not "write-through".

Bit 0x0010 on indicates I/O is "write-through".

Bit 0x0020 off indicates data for this I/O should be cached.

Bit 0x0020 on indicates data for this I/O should not be cached.

All other bits are reserved are zero.

The difference between the "cache bypass" and the "no cache" bits is in the type of request packet that the device driver will be passed. With "cache bypass", it will get a packet with command code 24, 25, or 26. With "no cache", the system gets the extended packets for command codes 4, 8, or 9.

hVPB volume handle for source of I/O

pData long address of user transfer area

pcSec pointer to number of sectors to be transferred. On return this is the number of sectors successfully transferred.

iSec sector number of first sector of transfer

Returns Error code if operation failed, 0 otherwise.

ERROR_PROTECTION_VIOLATION - the supplied address/length is not valid.

ERROR_UNCERTAIN_MEDIA - the device driver can no longer reliably tell if the media has been changed. This occurs only within the context of an FS_MOUNT call.

ERROR_TRANSFER_TOO_LONG - transfer is too long for device

FSH_DOVOLIO may be used at all times within an FSD. When called within the scope of a FS_MOUNT call, it applies to the volume in the drive without regard to which volume it may be. However, since volume recognition is not complete until the FSD returns to the FS_MOUNT call, the FSD must take special precautions when an ERROR_UNCERTAIN_MEDIA is returned. This indicates that the media has gone uncertain trying to identify the media in a drive. This may indicate that the volume that the FSD was trying to recognize was removed. In this case, an FSD releases any resources attached to the hVPB passed in the FS_MOUNT call and returns ERROR_UNCERTAIN_MEDIA to the FS_MOUNT call. This will direct the volume tracking logic to restart the mount process.

FSDs call FSH_DOVOLIO2 to control device driver operation independently from I/O operations. This routine supports volume management for IOCTL operations. Any errors are reported to the hard error daemon before returning to the FSD. Any retries indicated by the hard error daemon or actions indicated by DOSERROR are done within the call to FSH_DOVOLIO2.

```

15      int far pascal FSH_DOVOLIO2 (hDev, sfn, cat, func, pParm, cbParm, pData, cbData)
      unsigned long hDev;
      unsigned short sfn;
      unsigned short cat;
      unsigned short func;
      char far *    pParm;
20      unsigned short cbParm;
      char far *    pData;
      unsigned short cbData;

```

Where:

25 hDev device handle obtained from VPB

sfn system file number from open instance that caused the FSH_DEVIOCTL call. This field should be passed unchanged from the sfi selfsfn field. If no open instance corresponds to this call, this field is set to 0xFFFF.

cat category of IOCTL to perform

30 func function within category of IOCTL

pParm long address to parameter area

cbParm length of parameter area

pData long address to data area

cbData length of data area

35 Returns Error code if error detected, 0 otherwise.

The ERROR_INVALID_FUNCTION is invoked when a function supplied is incompatible with the system of the present mention. It allocates a new VPB whenever the media becomes uncertain (the device driver recognizes that it can no longer be certain that the media is unchanged). This VPB cannot be collapsed with a previously allocated VPB (due to a reinsertion of media) until the FS_MOUNT call returns.

40 However, the previous VPB may have some cached data that must be updated from the media (the media may have been written while it was removed). FSH_FINDDUPHVPB allows the FSD to find this previous occurrence of the volume in order to update the cached information for the old VPB. The newly created VPB is unmounted if there is another, older VPB for that volume.

The calling format for FSH_FINDDUPHVPB is as follows.

```

45      int far pascal FSH_FINDDUPHVPB (hVPB, phVPB)
      unsigned short hVPB;
      unsigned short far * phVPB;

```

50 Where:

hVPB handle to the volume to be found

phVPB pointer to where handle of matching volume will be stored.

Returns Error code if no matching VPB found. 0 otherwise.

55 ERROR_NO_ITEMS - there is no matching hVPB.

FSH_GETVOLPARM allows an FSD to retrieve file-system-independent and -dependent data from a VPB. Since the FS router passes in a VPB handle, individual FSDs map the handle into pointers to the relevant portions. The calling sequence for FSH_GETVOLPARM is as follows:

```

void far pascal FSH_GETVOLPARM (hVPB, ppVPBfsi, ppVPBfsd)
unsigned short hVPB;
struct vpfsi far * far * ppVPBfsi;
struct vpfsd far * far * ppVPBfsd;

```

5

Where:

hVPB volume handle of interest

ppVPBfsi location of where pointer to file-system- independent data is stored

10 ppVPBfsd location of where pointer to file-system- dependent data is stored

Returns: Nothing

Because FSD-Volume mapping is dynamic, and FSD-DD connections are achieved through the operating system kernel in an FSD and DD independent way, any FSD may access any volume, including volumes whose DDs were loaded from that FSD. Since a volume maps to a particular piece of removable media or to any partition on any partitionable media, it is contemplated that multiple FSDs may have access to a particular hard disk or other media.

Volume file operations are divided into two categories: Named-based operations and handle-based operations. Name-based operations are typically initiated by a user wherein a user instructs the system 100 to perform a named operation on a file. Handle-based operations are typically initiated during the background operation of the system. Handle-based operations are usually preceded by a name-based operation.

Referring now to Figure 8, the routine 800 is invoked when the system 100 performs named-based operations. A named operation is an operation which is directed by a character name, i.e. the operation is specified with the name of a file or directory. "Open file 'xxx'" is one example of a name-based operation. Process 802 is invoked to parse the name and return three variables: PathNameType; TCBThisVPB and TCBThisFSC. Process 802 is discussed in detail in conjunction with Figure 9. (Note: h denotes a handle and TCB refers to a thread control block wherein TCHThisVPB is handle to the VPB currently of interest and TCBThisFCH is the pointer to the file system of interest). Item 804 then routes control to the appropriate function based on the variables PathType, TchThisVPB and TCBThisFCH returned by process 802. Control is passed item 806 if the path began with "\\" indicating a Universal Naming Convention (UNC) global network name in which the UNC FSD is invoked. If a local device is indicated, control passes to item 808 to process the request within the kernel. If a pseudodevice or remote file is indicated, control passes to item 810 to route the request to the remote FSD to which the pseudodevice or remote file is attached. If a named pipe is detected, control passes to item 812 to call the local named pipe code within the kernel. If a local file is indicated, control passes to item 814 which is the FSD worker in the FSD which performs reads and writes to the volume by calling FSHDOVOLIO in item 816. FSHDOVOLIO is discussed further in conjunction with Figure 11.

Referring now to Figure 9, the parsing process 802 is described. When invoked, item 902 transforms the name of interest to a canonical form based on current drive, current directory and the name itself. The variables TCBTHISFSC and TCHThisVPB and pathnametype are then determined as follows. Decision 904 determines whether the user name begins with "\\" to determine whether a UNC name is indicated. If so, control passes to item 905, wherein the values of the variables PathType, TchThisVPB and TCBThisFCH are initialized to route the user name to the appropriate location. If not, decision 906 determines whether the name of interest is a name in the device name list maintained by the kernel. If so, decision 908 determines whether it is a pseudo-character device. If so, item 910 sets the variables as indicated. If not, control passes to item 912 which sets the variables as indicated.

Decision 914 determines whether the name represents a named pipe by looking for "pipe\\" at the beginning of the name. If so, item 916 sets the variables as indicated. If not, decision 918 determines whether the name indicates a pathname on a local or remote drive. If a remote drive is indicated, control passes to item 920 which sets the variables PathType, TchThisVPB and TCBThisFCH as indicated. Otherwise, control passes to item 922 which calls what volume to read the appropriate data from the volume. When WhatVolume returns, control passes to item 924 which sets the variables PathType, TchThisVPB and TCBThisFCH as indicated.

Referring now to Figure 10, the process 1000 is invoked for handle-based operations. When invoked, item 1002 retrieves an SFT entry. The SFT entry and the handle are both set up by DosOpen. TCBThisFSC is then set as indicated. Item 1004 then calls the relevant FSD worker for the file system that FSC points to. The hVPB is passed along from the SFT entry. Item 1006 then calls item 1016 to perform any I/O requested by the caller by calling item 1016 as required.

Referring now to Figure 11, FSH Do Vol IO is shown. When invoked in item 1102 the hVPB is used to determine what volume is in the drive as well as the volume of interest. Decision 1104 then determines whether the volume in the drive is the volume of interest. If so, 1106 is invoked to call the device driver and to perform I/O with the parameters specified. Decision 1108 then determines whether the media went uncertain during the operation. If not, the process returns in item 1114. If decision 1108 determines the media is not uncertain, control passes to item 1112 where WhatVolume is invoked to make the media certain. Control then returns to decision 1104. If the volume in the drive does not match the volume of interest, item 1110 is invoked to call HardError to instruct the user to place the correct volume in the drive. Control then passes to item 1112 described above.

- 10 Appendices II - VI are included herewith as an example of an installable file system source where:
 Appendix II is a listing of exported interfaces a file system is expected to support in accordance with the teachings of the present invention.
 Appendix III is a listing of interfaces exported by a kernel which a file system may use.
 Appendix IV is the source code of an example of an installable file system constructed in accordance with the present invention.
 15 Appendix V is a listing of a definitions file used by the OS/2 linker to build the FSD of Appendix IV.
 Appendix VI is a header file that defines structures and parameters used by the IFS of Appendix IV.

In summary, an improved method and system for dynamic volume tracking in an installable file system has been described. In accordance with the teachings of the present invention, a computer system is provided with a plurality of file system drivers which are organized in a linked list. The system is further provided with a default file system driver. Whenever a new volume is presented to the system or whenever media becomes uncertain, the system automatically and dynamically invokes the file system drivers until a file system driver accepts the volume. If no file system driver accepts the volume, the default file system is mounted. Accordingly, other uses and modifications will be apparent to persons of ordinary skill in the art.
 20 All such uses and modifications are intended to fall within the spirit and scope of the appended claims.
 25

Claims

- 30 1. A method for mounting a file system for use in communicating with a data storage device and a computer system comprising the steps of:
 a) including a plurality of file system modules in a computer operating system including a default file system wherein said file systems are organized in a linked sequence;
 b) coupling a data storage device to said computer system;
 35 c) detecting a change in media in said data storage device or the first time said computer system accesses said data storage device;
 d) loading a file system identified in said list of file systems;
 e) reading a volume identifier from said media wherein the location of said volume identifier is specified by said loaded file system;
 40 f) comparing the read volume identifier from said media with the identifier associated with said file system;
 g) mounting said file system if said identifiers match;
 h) loading the next file system identified in said list of file systems if said identifiers do not match;
 i) returning to step (e) until each file system in said list of file systems has been tested or until a match is found; and
 45 j) mounting a default file system if no match is found.

50

55

Appendix I

<p><</p> <p>OS/2 1.2 IFS Patent Documentation</p> <p>08-24-1989</p> <p>OS/2 Systems Division Microsoft Corp. 16011 NE 36th Way Box 97017 Redmond, WA 98073-9717</p> <p>This document contains information of a proprietary nature. All information contained herein shall be kept in confidence. None of this information shall be divulged to persons other than Microsoft employees authorized by the nature of their duties to receive such information. No part of this document shall be reproduced or distributed without permission of the originator.</p>	<p>Microsoft Confidential OS/2 1.2 IFS Patent Documentation</p> <p>PREFACE</p> <p>Preface</p> <p>11</p>
--	---

EP 0 415 346 A2

Contents

Preface	11
1.0 GENERAL OBJECTIVES	1
1.0.1 Summary of Functional Characteristics	1
1.0.1.1 Improved User Interface	1
1.0.1.2 Installable File System Mechanism	1
1.0.1.3 Long Name Support	1
1.0.1.3.1 Meta Character Semantics	2
1.0.1.3.2 Device Names	4
1.0.1.3.3 General Utility Changes	4
1.0.1.3.4 Long Names for Non-file System Objectives	7
2.0 Functional Characteristics	9
2.1 Programming Interface	9
2.1.0.1 File System Enhancements	9
2.1.0.1.1 GET/SET EXTENDED ATTR. (INT 21H, 5/02H and 5703H)	9
2.1.0.1.2 ENUMERATE EXTENDED ATTRIBUTES (INT 21H, 6EH)	12
2.1.0.1.3 QUERY FILE SYSTEM FOR MAX EA SIZE (INT 21H, 6FH)	15
2.1.0.1.4 EXTENDED OPEN/CREATE (INT 21H, 6CH)	15
2.1.0.1.5 GET/SET MEDIA ID (INT 21H, 69H)	19
2.1.0.2 Miscellaneous Enhancements	21
2.1.0.2.1 QUERY DOS VALUE (INT 21H, 3305H)	21
2.1.1 Interprocess Communication	22
2.1.1.1 Communication via flags	22
2.1.1.2 Communication via messages	22
2.1.1.2.1 Pipes	22
2.1.1.2.2 Queues	23
2.1.1.2.3 Comparing Pipes and Queues	23
2.1.1.3 Coordinating execution among several threads	24
2.1.1.3.1 Semaphores	24
2.1.1.3.2 Starting and Stopping a Thread's Execution	25
2.1.1.3.3 DosFlagProcess - Set a Process's External Event Flag	25
2.1.1.4 Pipes	27
2.1.1.4.1 DosMakePipe - Create a Pipe	28
2.1.2 Named Pipes	29
2.1.2.1 Named Pipe Function Calls	31
2.1.2.1.1 DosCallNmPipe:- Perform a procedure call via a message pipe.	31
2.1.2.1.2 DosConnectNmPipe:- Waits for a new client to open a pipe.	32
2.1.2.1.3 DosDisconnectNmPipe:- Forces a named pipe closed.	33
2.1.2.1.4 DosMakeNmPipe - Create a named pipe.	34
2.1.2.1.5 DosPeekNmPipe:- Peek into a named pipe.	40
2.1.2.1.6 DosQNmPipeHandState - Query Named Pipe Handle State	43
2.1.2.1.7 DosQNmPipeInfo - Query a named pipe's information	45
2.1.2.1.8 DosRawReadNmPipe:- Direct (raw) read named pipe	46
2.1.2.1.9 DosRawWriteNmPipe:- Direct (raw) write named pipe	47
2.1.2.1.10 DosSetNmPipeHandState - Set Named Pipe Handle State	48

Contents

2.1.2.1.11 DosTransactNmPipe:- Perform a transaction on a message pipe.	51
2.1.2.1.12 DosWaitNmPipe:- Wait for an available named pipe instance.	51
2.1.3 I/O Services	52
2.1.4 ANSI Support for Video and Keyboard	53
2.1.4.1 Device and File Handles	53
2.1.4.2 Handling of I/O	53
2.1.4.3 ASCII Strings	54
2.1.4.4 Filename Specification	55
2.1.4.5 File Allocation Table (FAT) Type Determination	56
2.1.4.6 Device Names	57
2.1.4.7 Device I/O Services - General API	58
2.1.4.7.1 DosBeep - Generate Sound From Speaker	58
2.1.4.7.2 DosDevIOCtrl - I/O Control for Devices	58
2.1.5 Extended DOS Partition Architecture	59
2.1.5.1 Installing Block Devices in the Extended Partition	61
2.1.5.2 Creating Block Devices in the Extended DOS Partition	62
2.1.5.3 Deleting Block Devices in the Extended DOS Partition	63
2.1.5.4 Layout of Block Devices in the Extended DOS Partition	63
2.1.5.5 BPB and Get Device Parameters for Extended Volumes	66
2.1.5.6 Category 8 Generic IOCTL Commands	66
2.1.5.7 Type 6 Partition	66
2.1.5.8 Layout of Block Devices With A Type 6 Partition	68
2.1.5.9 Layout of Block Devices With A Type 6 Partition	69
2.1.5.9.1 Type 7 Partition	69
2.1.6 Code page support	71
2.1.6.1 Code Page API	71
2.1.6.1.1 DosSetCp - Set Code Page	71
2.1.6.1.2 DosSetProcCp - Set Process Code Page	73
2.1.6.1.3 DosGetCp - Get Process Code Page	74
2.1.6.1.4 DosGetCtryInfo - Get Country Information	75
2.1.6.1.5 DosCaseMap - Get Case Mapping	77
2.1.6.1.6 DosGetDBCSEv - Get DBCS Environment	78
2.1.6.1.7 DosGetCollate - Get Collating Sequence	80
2.1.7 System Initialization and Configuration	82
2.1.7.1 System Initialization	82
2.1.7.1.1 The Boot Sector	82
2.1.7.1.2 OS/2 (TM) Device Interface Module	82
2.1.7.1.3 OS/2 (TM) Kernel Module	83
2.1.7.1.4 OS/2 (TM) System Initialization Process	83
2.1.7.2 System Configuration (CONFIG.SYS)	83
2.1.7.3 OS/2 (TM) Configuration Command Descriptions	84
2.1.7.3.1 AUTOFAIL	84
2.1.7.3.2 BUFFERS	84
2.1.7.3.3 COUNTRY	85
2.1.7.3.4 DEVICE	85
2.1.8 Installable File System	86
2.1.8.1 Requirements on The IFS Mechanism	86

Contents

2.1.8.2 Description	87
2.1.8.2.1 System Relationships	87
2.1.8.2.2 Standard File I/O	89
2.1.8.2.3 Extended File I/O	90
2.1.8.2.4 I/O	91
2.1.8.2.5 Buffer Management	94
2.1.8.2.6 Volume Management	94
2.1.8.2.7 Connectivity	95
2.1.8.2.8 IPL Mechanism	95
2.1.8.2.9 OS/2 Partition Access	96
2.1.8.2.10 Permissions	96
2.1.8.2.11 File Naming Conventions	96
2.1.8.2.12 Meta Character Processing	98
2.1.8.2.13 Family API Issues	99
2.1.8.2.14 FS Utility Support	100
2.1.8.2.15 FSD Pseudo-Character Device Support	101
2.1.8.3 Extended Attributes	101
2.1.8.3.1 FSD File Image	104
2.1.8.3.2 FSD Attribute, FS_ATTRIBUTE	105
2.1.8.4 FSD Initialization	106
2.1.8.4.1 OS/2 and PC/DOS 3.4 Boot Record and BIOS Parameter Block	107
2.1.8.5 IFS Commands	108
2.1.8.5.1 IFS CONFIG.SYS Function	108
2.1.8.6 IFS API Function Calls	109
2.1.8.6.1 Application File I/O Notes	111
2.1.8.6.2 Application File I/O Function Calls	113
2.1.8.6.3 DosBufReset - Commit file's cache buffers	113
2.1.8.6.4 DosChDir - Change The Current Directory	114
2.1.8.6.5 DosChgFilePtr - Change File Read Write Pointer	115
2.1.8.6.6 DosClose - Close a File Handle	117
2.1.8.6.7 DosCopy - Copy file	118
2.1.8.6.8 DosDelete - Delete a File	122
2.1.8.6.9 DosDevIOCtrl - I/O Control for Devices	124
2.1.8.6.10 DosDupHandle - Duplicate a File Handle	126
2.1.8.6.11 DosEditName - Transform source string using editing strings	128
2.1.8.6.12 DosEnumAttribute - Enumerate the extended attributes	129
2.1.8.6.13 DosFileIO - File I/O	132
2.1.8.6.14 DosFileLocks - File Lock Manager	136
2.1.8.6.15 DosFindClose - Close Find Handle	138
2.1.8.6.16 DosFindFirst - Find First Matching File	139
2.1.8.6.17 DosFindNext - Find Next Matching File	145
2.1.8.6.18 DosFindNotifyClose - Close Find-Notify Handle	147
2.1.8.6.19 DosFindNotifyFirst - Start monitoring directory for changes	148
2.1.8.6.20 DosFindNotifyNext - Return directory	150
2.1.8.6.21 DosFsAttach - Creates and destroys FSD connections	152
2.1.8.6.22 DosFsCtl - File System Control	154
2.1.8.6.23 DosMkDir - Make Subdirectory	157

Contents

2.1.8.6.24 DosMove - Move a File or a Subdirectory	159
2.1.8.6.25 DosNewSize - Change File's Size	161
2.1.8.6.26 DosOpen - Open a File	162
2.1.8.6.27 DosQCurDir - Query Current Directory	170
2.1.8.6.28 DosQCurDisk - Query Current Disk	172
2.1.8.6.29 DosQFHandState - Query File Handle State	173
2.1.8.6.30 DosQFileInfo - Query a File's Information	176
2.1.8.6.31 DosQFileMode - Query File Mode	179
2.1.8.6.32 DosQFsAttach - Query attached FSD information	181
2.1.8.6.33 DosQFsInfo - Query File System Information	184
2.1.8.6.34 DosQHandType - Query a Handle Type	186
2.1.8.6.35 DosQPathInfo - Query a file or a subdirectory for information	186
2.1.8.6.36 DosQSysInfo - Query System Information	192
2.1.8.6.37 DosQVerify - Query Verify Setting	193
2.1.8.6.38 DosRead - Read from a File	194
2.1.8.6.39 DosReadAsync - Async Read from a File	196
2.1.8.6.40 DosRmdir - Remove Subdirectory	198
2.1.8.6.41 DosScanEnv - Scan Environment Segment	199
2.1.8.6.42 DosSearchPath - Search a path for a file name	200
2.1.8.6.43 DosSelectDisk - Select Default Drive	203
2.1.8.6.44 DosSetFHandState - Set File Handle State	204
2.1.8.6.45 DosSetFileInfo - Set a File's Information	207
2.1.8.6.46 DosSetFileMode - Set File Mode	209
2.1.8.6.47 DosSetFsInfo - Set File System Information	211
2.1.8.6.48 DosSetMaxFH - Set Maximum File Handles	213
2.1.8.6.49 DosSetPathInfo - Set a File's or Directory's Information	213
2.1.8.6.50 DosSetVerify - Set/Reset Verify Switch	217
2.1.8.6.51 DosShutdown - Shutdown File Systems for Power Off	218
2.1.8.6.52 DosWrite - Synchronous Write to a File	219
2.1.8.6.53 DosWriteAsync - Asynchronous Write to a File	221
2.1.8.7 FSD System Interfaces	222
2.1.8.8 Overview of Installable File System Driver Dispatch	222
2.1.8.9 Data Structures	223
2.1.8.9.1 Time Stamping	226
2.1.8.10 FSD calling conventions and requirements	226
2.1.8.11 Error codes	227
2.1.8.11.1 FS service routine command names	227
2.1.8.11.2 FS Entry Point Descriptions	228
2.1.8.11.3 FS_ATTACH - Attach or Detach An FSD To A Drive or Device	233
2.1.8.11.4 FS_CHDIR - Change/Verify Directory Path	233
2.1.8.11.5 FS_CHGFILEPTR - Move a file's position pointer	237
2.1.8.11.6 FS_CLOSE - Close a File	239
2.1.8.11.7 FS_COMMIT - Commit a file's buffers to Disk	241
2.1.8.11.8 FS_COPY - Copy a file	243
2.1.8.11.9 FS_DELETE - Delete a File	245
2.1.8.11.10 FS_EXIT - End of process	246
2.1.8.11.11 FS_FILEATTRIBUTE - Query/Set File Attribute	247

Contents

2.1.8.11.12	FS_FILEINFO - Query/Set a File's Information	248
2.1.8.11.13	FS_FILEIO - Multi-function File I/O	250
2.1.8.11.14	FS_FINDCLOSE - Directory Read (Search) Close	253
2.1.8.11.15	FS_FINDFIRST - Find First Matching File Name	254
2.1.8.11.16	FS_FINDFROMNAME - Find Matching File Name Starting from Name	258
2.1.8.11.17	FS_FINDNEXT - Find Next Matching File Name	260
2.1.8.11.18	FS_FINDNOTIFYCLOSE - Close Find-Notify Handle	262
2.1.8.11.19	FS_FINDNOTIFYFIRST - Start monitoring directory for changes	263
2.1.8.11.20	FS_FINDNOTIFYNEXT - Resume reporting directory changes	265
2.1.8.11.21	FS_FLUSHBUF - Commit file buffers	267
2.1.8.11.22	FS_FSETL - File System Control	268
2.1.8.11.23	FS_FINFO - File System Information	271
2.1.8.11.24	FS_INIT - File system driver initialization	272
2.1.8.11.25	FS_IOCTL - I/O Control for Devices	273
2.1.8.11.26	FS_MKDIR - Make Subdirectory	274
2.1.8.11.27	FS_MOUNT - Mount/Unmount volumes	275
2.1.8.11.28	FS_MOVE - Move a File or Subdirectory	278
2.1.8.11.29	FS_NEWSIZE - Change File's Logical Size	280
2.1.8.11.30	FS_NMPIPE - Do a remote named pipe operation	281
2.1.8.11.31	FS_OPENCREATE - Open a File	285
2.1.8.11.32	FS_PATHINFO - Query/Set a File's Information	288
2.1.8.11.33	FS_PROCESSNAME - Allow FSD to modify name after OS/2	290
2.1.8.11.34	FS_READ - Read from a File	291
2.1.8.11.35	FS_RMDIR - Remove Subdirectory	293
2.1.8.11.36	FS_SETSWAP - Notification of swap-file ownership	294
2.1.8.11.37	FS_SHUTDOWN - Shutdown File System for Power Off	295
2.1.8.11.38	FS_WRITE - Write to a File	297
2.1.8.12	FS Helper Functions	298
2.1.8.12.1	FS Services Supplied by OS/2	298
2.1.8.12.2	FS Help Routines	298
2.1.8.12.3	FSH_ADDSHARE - Add a name to the share set	301
2.1.8.12.4	FSH_HUPSTATE - Get or set buffer state	303
2.1.8.12.5	FSH_CANONICALIZE - Convert pathname to canonical form	304
2.1.8.12.6	FSH_CHECKENAME - Check extended attribute name validity	305
2.1.8.12.7	FSH_CRITERION - Signal a hard error to the daemon	306
2.1.8.12.8	FSH_DEVIOTL - Send IOCTL request to device driver	308
2.1.8.12.9	FSH_DOVOLIO - Volume-based sector-oriented transfer	310
2.1.8.12.10	FSH_DOVOLIO2 - Send volume-based IOCTL request to device driver	311
2.1.8.12.11	FSH_FINDCHAR - Find first occurrence of char in string	315
2.1.8.12.12	FSH_FINDUPHVPB - Locate equivalent hVPB	316
2.1.8.12.13	FSH_FLUSHBUF - Flush buffered data to a volume	317
2.1.8.12.14	FSH_FORCENOSWAP - Force segments permanently into memory	318
2.1.8.12.15	FSH_GETBUF - Buffered sector read	319
2.1.8.12.16	FSH_GETFIRSTOVERLAPBUF - Locates buffer overlapping range	321
2.1.8.12.17	FSH_GETVOLPARM - Get VPB data from VPB handle	323
2.1.8.12.18	FSH_INTERR - Signal an internal error	324
2.1.8.12.19	FSH_ISCURDIRPREFIX - Test for a prefix of a current directory	325

Contents

2.1.8.12.20	FSH_LOADCHAR - Load a character from a string	326
2.1.8.12.21	FSH_NAMEFROMSFN - Get the full path name from an SFN	327
2.1.8.12.22	FSH_PREVCHAR - Decrement character pointer	328
2.1.8.12.23	FSH_PROBEBUF - User address validity check	329
2.1.8.12.24	FSH_QSYSINFO - Query system information	331
2.1.8.12.25	FSH_RELEASEBUF - Release the owned buffer	332
2.1.8.12.26	FSH_REMOVESHARE - Remove a share entry	333
2.1.8.12.27	FSH_SEGALLOC - Allocate a GDT or LDT segment	334
2.1.8.12.28	FSH_SEGFREE - Release a GDT or LDT segment	336
2.1.8.12.29	FSH_SEGREALLOC - Change segment size	337
2.1.8.12.30	FSH_SEMCLER - Clear a semaphore	338
2.1.8.12.31	FSH_SEMREQUEST - Request a semaphore	339
2.1.8.12.32	FSH_SEMSET - Set a semaphore	340
2.1.8.12.33	FSH_SEMSETWAIT - Set a semaphore and wait for clear	341
2.1.8.12.34	FSH_SEMWAIT - Wait for clear	342
2.1.8.12.35	FSH_STORECHAR - Store a character in a string	343
2.1.8.12.36	FSH_UPPERCASE - Uppercase ascii string	344
2.1.8.12.37	FSH_WILDMATCH - Match using OS/2 wildcards	345
2.1.8.12.38	FSH_YIELD - Yield CPU to higher priority threads	346
2.1.8.13	Overview	346
2.1.8.14	Operational Description	347
2.1.8.15	FAT Boot Procedure	347
2.1.8.16	Non-FAT Boot Procedure	348
2.1.8.17	Interfaces	349
2.1.8.18	Black Box to Mini-FSD Interface	351
2.1.8.19	OS2LDR Interface	351
2.1.8.20	Stage 1 Interfaces	352
2.1.8.20.1	MFS_CHGFILEPTR	353
2.1.8.20.2	MFS_CLOSE	353
2.1.8.20.3	MFS_INIT	353
2.1.8.20.4	MFS_OPEN	355
2.1.8.20.5	MFS_READ	355
2.1.8.20.6	MFSH_DOVOLIO	356
2.1.8.20.7	MFSH_INTERR	356
2.1.8.20.8	MFSH_SEGALLOC	357
2.1.8.20.9	MFSH_SEGFREE	357
2.1.8.20.10	MFSH_SEGREALLOC	358
2.1.8.21	Stage 2 Interfaces	358
2.1.8.22	Stage 3 Interfaces	359
2.1.8.22.1	MFS_TERM	359
2.1.8.23	Imbedded Device Driver Helpers	360
2.1.8.23.1	MFSH_CALLRM	360
2.1.8.23.2	MFSH_LOCK	361
2.1.8.23.3	MFSH_PHYSIOVIRT	361
2.1.8.23.4	MFSH_UNLOCK	362
2.1.8.23.5	MFSH_UNPHYSIOVIRT	362
2.1.8.23.6	MFSH_VIRT2PHYS	363

Microsoft Confidential
OS/2 1.2 IFS Patent Documentation

Contents

2.1.8.24 Special Considerations	363
2.1.8.25 Constraints	363
2.1.8.26 Limitations	364
2.1.8.27 Dependencies	364
Index.....	366

Microsoft Confidential
OS/2 1.2 IFS Patent Documentation

List of Illustrations

Figure 1. Example of layout of large DASD device with Extended DOS partition.	
Figure 2. Example of layout of large DASD device with a type 6 partition.	
Figure 3. Example of layout of large DASD device with type 6 partition.	60
Figure 4. System Relationships for Installable File Systems	88
Figure 5. Standard File I/O	89
Figure 6. Extended File I/O	90
Figure 7. Synchronous I/O Example	92
Figure 8. Asynchronous I/O Example	93
* Figure 9. DOSFINDFIRST return buffer format	142

1.0 GENERAL OBJECTIVES

1.0.1 Summary of Functional Characteristics

1.0.1.1 Improved User Interface

- The user interface is greatly improved to include the capability to run and view multiple programs at the same time. For additional information see the Presentation Manager specification.

1.0.1.2 Installable File System Mechanism

- The Installable File System will enable users to replace the present file system. AI based, with a file system of their choice. The user may choose the file system which meets their needs. This will provide the user with the capability of utilizing a greater disk capacity, additional file flexibility, or the potential for increased performance. They can have multiple active file systems on a single PC, with the capability of multiple and different storage devices. And there is support for multiple logical volumes (partitions).

1.0.1.3 Long Name Support

- Long Name support means that a program (like our utilities or command line processors) must understand that:

- ASC112 string representing fully qualified file specifications (i.e. drive letter, path, and file name) may now be up to 260 bytes in length, and may get bigger in a future release.

- Note: MAXPATHLEN defined in Doscalls.Hwc and DosQSysinfo can be used to determine the maximum path length.

- ASC112 string representing a file name component may be 255 bytes in length.

1.0 GENERAL OBJECTIVES

Any program which assumes a file name component of 13 bytes must be changed. Long File Names will only be supported in protect mode. The FAT file system will continue to only support 8.3 names. While spaces are valid file name characters, our utilities will continue to not support them.

Note: MAXCOMPLEN can be used to determine the maximum file name component length. The structures in Doscalls.Hwc will be changed to use MAXCOMPLEN instead of 13.

The file system defines the component separator within a file name to be '.'. However, there is no limit on the number of components which may be allowed within a file name. Any program which is counting on only finding one period within a file name must be changed.

The file system allows the component separator within a path name to be '/' or '\'. Our utilities will continue to use the convention that '\' is the path name component separator and '/' is the switch character.

Components and other affected areas:

CMD.EXE
ATTRIB
BACKUP
CHKDSK
COMP
FIND
PATCH
PRINT
RECOVER
REPLACE
RESTORE
The Spooler
TREE
XCOPY
DOSCALLS.HWC
System Initialization
VIO and KBD subsystems and KEYB utility
NIS
Session Manager
Shell
Swapper
Message Retriever
DosSearchPath

1.0.1.3.1 Meta Character Semantics:

Use DosFindFirst/DosFindNext to determine all source files which match the wildcard name. (This is probably not a change for any program.)

The file system will provide a new API to transform a source file name

1.0 GENERAL OBJECTIVES

into a destination file name for a given editing string. All utilities which must perform name conversion (like copy or rename) should change to use this new API. The API is called DosEditName.

Any program which must convert a file specification to its fully qualified form (i.e. drive letter, path, file name) should use DosQPathInfo Level 5.

Any program which currently defaults the source file name to *.* like DIR and ERASE must change the default to *.*.

CMD.EXE and COMMAND.COM must prompt the user for verification when he specified *DEL ** for any file system or *DEL *.* for the FAT file system.

DosEditName will have the following interface:

```
extern unsigned far pascal DosEditName (  
    unsigned EditLevel,  
    char far *SourceString,  
    char far *EditString,  
    char far *TargetBuf,  
    unsigned TargetBufLen  
);
```

The meanings of the fields are defined below:

EditLevel Which semantics to use. 1 is currently the only defined level.

SourceString A file name component with a maximum length equal to MAXCOMPLEN that contains no meta characters, no path characters, and no drive specification.

EditString A file name component with a maximum length equal to MAXCOMPLEN that can contain meta characters, but no path characters and no drive specification.

TargetBuf Where the resultant file name is stored

TargetBufLen The size of TargetBuf. This is limited to MAXCOMPLEN.

Components and other affected areas:

CMD.EXE
ATTRIB
BACKUP
CHKDSK
COMP
PRINT
REPLACE
RESTORE
XCOPY

1.0 GENERAL OBJECTIVES

3

RECOVER

1.0.1.3.2 Device Names:

1.0.1.3.2.1 3xBox:

No change from OS/2 1.1 and PC-DOS except that device names with bogus paths will now be accepted.

1.0.1.3.2.2 Protect Mode:

Paths and extensions are SIGNIFICANT on device names in protect mode as are trailing dots. This means that "C:\DEVICE" is different from "DEVICE." which is different from "DEVICE". This also means that 'NUL.LST' will create a file called 'NUL.LST'.

The following set of device names, as an exception, are said to exist in all directories: CON, KBD\$, MOUSE\$, COM1 thru COM9, LPT1 thru LPT9, PRN, SCREEN\$, NUL, POINTER\$, CLOCK\$.

All other devices, whether installed or pseudo, exist only in the directory \DEV\, and thus may only be reached with a path of the form \DEV\DEVNAME. What this also means is that if you attempt to fully qualify a device name, using DosQPathInfo level 5, it will get a \DEV\ prepended to it.

1.0.1.3.3 General Utility Changes:

1.0.1.3.3.1 Dir Command:

* FAT filesystem and /N, /W, and /F not specified

Display exactly as today: edited file name, file size data, last update date, last update time.

* Non FAT filesystem or /N specified

- /W switch not specified

Last update date, last update time, file size data, file size extended attributes, unedited file name

- /W switch is specified

The list of found files is pre-scanned to find the longest file name. The width of the screen is divided by the longest file name to determine the number of files that will be displayed per line. Unedited filenames are displayed in columnar format.

* /F switch

Displays the fully qualified file name of each matching file. No

1.0 GENERAL OBJECTIVES

4

00007

heading or trailing information is displayed.

1.0.1.3.3.2 Searching For Executables:

This section describes the changes for supporting Searching For Executables in the Long File Name environment.

The old method was as follows:

1. If a '.' existed in the file name component, strip of the '.' and any remaining characters.
2. In the current directory and for each path in PATH attempt to find the file with an added extension of '.COM', '.EXE', '.CMD'.

The new method is as follows:

1. If the extension is not empty, i.e. there is at least one '.' in the file name and what follows the last '.' is not NULL look for the file as specified.
2. If the extension is '.CMD' or '.BAT' execute it as a batch file.
3. Anything else is treated as an executable file
4. If the extension was empty or file not found, then in order append, '.COM', '.EXE', '.CMD' and look for file.
5. If not in current directory, repeat steps 1 thru 4 for all directories in path.

The following examples will illustrate the search order.

FOO FOO.COM, FOO.EXE, FOO.CMD
FOO. FOO..COM, FOO..EXE, FOO..CMD
FOO.EX1 FOO.EX1, FOO.EX1.COM, FOO.EX1.EXE, FOO.EX1.CMD
FOO.BAT FOO.BAT, FOO.BAT.COM, FOO.BAT.EXE, FOO.BAT.CMD
FOO.EXE FOO.EXE, FOO.EXE.COM, FOO.EXE.EXE, FOO.EXE.CMD

1.0.1.3.3.3 IFS versions of CHKDSK, FORMAT, RECOVER, SYS:

- * Only have to run in protect mode
- * Will be contained in a file called Uxxxxxxx.DLL, where xxxxxxx is the exported name of the FSD.

xxxxxxx should be (up to) the first 7 characters of the exported FSD name, since we do not limit the name to 7 (or less) characters.

- * The entry points within Uxxxxxxx.DLL will be the name of the utility whose function they provide.

1.0.1.3.3.4 DISKCOMP and DISKCOPY:

Will continue to handle diskettes written by any file system. Document strange case under which DISKCOPY corrupts data because it thinks it is the volume serial number and DISKCOMP would not notice a mismatch when working with IFS formatted diskettes.

1.0.1.3.3.5 FORMAT:

Will use the following algorithm to determine how to format the volume:

- * If the user specified /FS:xxxxxxx then format the volume for the specified file system.
- * If the volume is already formatted, reformat it for the same file system.
- * If in protect mode and there exists an environment variable for FORMAT-y where y is /FS:xxxxxxx, format the volume for the specified file system.
- * Format the volume for the FAT file system.
- * If FORMAT was started in the real mode environment, and it would otherwise format for a file system other than FAT, reject the request.

1.0.1.3.3.6 UTILAPI.LIB:

Provides mode-independent support without adding into FAPI (API.LIB) interfaces which are not supported by PC/DOS.

- * DosDevIoctl2 (performs DosDevIoctl function)
- * DosQFsAttach (returns FSD name)

1.0.1.3.3.7 System Installation:

If the user specifies to format the OS/2 partition for any file system other than FAT, add the statement "SET FORMAT=/FS:xxxxxxx" to the CONFIG.SYS file (xxxxxxx is the name of the selected file system).

1.0.1.3.4 Long Names For Non-file System Objectives:

File paths now allow up to 260 bytes (including trailing null byte). This same limit should apply to non-file system objects. This includes queue names, system semaphore names, shared memory names, and named pipe names.

1.0.1.3.4.1 Queue Names:

Queue names are currently limited to 128 bytes (by a local symbolic constant and several references to 128).

Queue names should be limited to the same length as other paths (260 bytes).

1.0.1.3.4.2 System Semaphore Names:

System Semaphore names are currently limited by the size of RMP buffers (260 bytes), but it stores two extra bytes, so the name portion is limited to 258 bytes (with trailing null).

System Semaphore names up to 260 bytes should be allowed (including the trailing null).

1.0.1.3.4.3 Shared Memory Names:

Shared Memory names are currently limited to 259 bytes (including trailing null), due to an off by one error (which did not appear prior to 1.2).

Shared Memory names will be fixed to allow up to 260 bytes.

1.0.1.3.4.4 Named Pipe names:

Named Pipe names were limited to 128 bytes in 1.1.

Named Pipe names will be updated to allow up to 260 bytes (already done).

1.0.1.3.4.5 Long Paths in the 3xBox:

The 3xBox should support a maximum of 260 bytes for paths. (Note that current directories are limited to 64 bytes in the 3xBox.) (Note also that filenames (components) which cannot be expressed as 8.3 are invisible in the 3xBox.)

1.0.1.3.4.6 Long Paths and Long Names in the Loader:

The loader will support long paths up to 260 bytes and long names up to 255 bytes (without trailing null).

Affected areas are DLL loading and DosExecPgm. For DLLs, the module name will continue to be the filename without the .DLL suffix.

2.0 FUNCTIONAL CHARACTERISTICS

2.1 PROGRAMMING INTERFACE

2.1.0.1 File System Enhancements

2.1.0.1.1 GET/SET EXTENDED ATTR. (INT 21H, 5702H and 5703H):

Purpose The Real Mode Box File Date/Time Interrupt 21H will support setting extended attributes as well as other information. To query Extended Attributes, a subset or all of the EA information for a file can be retrieved. To set Extended Attributes a list of one or more type codes with values is supplied. Additionally the FAPI interface supports info levels to get or set information via DOSQFILEINFO, DOSQPathInfo, DOSSetFileInfo, and DOSSetPathInfo. This will allow OS/2 utilities to have the function of these calls in real mode.

The INT21H, AX=5702H and 5703H API will be unpublished in both OS/2 and TUGBOAT.

Format Calling Sequence:

Get/Set Extended Attributes

For GET subset of Extended Attributes (AX=5703H, DX=3):

```
# MOV    AX,5703H      ; Get information
# MOV    BX,HANDLE     ; File handle
# MOV    DX,3          ; Get Extended Attributes subset
# LDS    SI,FileName   ; Filename (use if BX=0FFFFH)
# LES    DI,EABuf      ; Extended Attribute EAOP
# INT    21H
# JC     ERROR
```

For GET Extended Attributes, all (AX=5704H, DX=4):

```
# MOV    AX,5703H      ; GET EXTENDED ATTRIBUTES
# MOV    BX,HANDLE
# MOV    DX,4          ; Get Extended Attributes all
# LDS    SI,FileName   ; Filename (use if BX=0FFFFH)
# LES    DI,EABuf      ; Extended attr. EAOP structure
# INT    21H
# JC     ERROR
```

For SET (5704):

```
# MOV    AX,5704H      ; SET EXTENDED ATTRIBUTES
# MOV    BX,HANDLE
# MOV    DX,2          ; Set Extended Attributes
# LDS    SI,FileName   ; Filename (use if BX=0FFFFH)
# LES    DI,LIST       ; Extended attr. list
# INT    21H
# JC     ERROR
```

FAPI support for utilities

For GET Information (5703H):

```
# MOV    AX,5703H      ; Get information
# MOV    BX,HANDLE     ; File handle
# MOV    CX,BUFLen     ; Length of buffer
# MOV    DX,InfoLevel  ; Information level
# LDS    SI,FileName   ; Filename (use if BX=0FFFFH)
# LES    DI,BUFFER     ; Information buffer
# INT    21H
# JC     ERROR
```

For SET Information (5704H):

```
# MOV    AX,5704H      ; Get information
# MOV    BX,HANDLE     ; File handle
# MOV    CX,BUFLen     ; Length of buffer
# MOV    DX,InfoLevel  ; Information level
# LDS    SI,FileName   ; Filename (use if BX=0FFFFH)
# LES    DI,BUFFER     ; Information buffer
# INT    21H
# JC     ERROR
```

Where: HANDLE is the handle returned by a previous file open.

FileName is the ASCIIZ full path name of the file or subdirectory.

EABuf is an EAOP structure with the following format.

```
EAOP      struc          ; EAOP structure
fpGEAList dd      ?      ; dword pointer to a GEAList
fpFEAList dd      ?      ; dword pointer to a FEAList
offError  dw      ?      ; Error offset
EAOP      ends
```

where GEAList is a structure with the following format:

```
GEAList    struc          ;GEA list structure
ListSize   dw      ?      ;length of list
GEA        <?,?,?...>    ;packed set of GEAs
.          .              ; ...
GEAList    ends
```

where GEA is a structure with the following format:

```
GEA        struc          ;GEA structure
NameSize   db      ?      ;length of name
Name        db      ".....",0 ;asciiz name
GEA        ends
```

where FEAList is a structure with the following format:

```
FEAList    struc          ;FEA list structure
ListSize   dw      ?      ;length of list
FEA        <?,?,?...>    ;packed set of FEAs
.          .              ; ...
FEAList    ends
```

where FEA is a structure with the following format:

```
FEA        struc          ;FEA structure
reserved   db      0      ;reserved must be zero
NameSize   db      ?      ;length of name
ValueSize   db      ?      ;length of value
Name        db      "...",0 ;asciiz name
Value       db      ?,?... ;free-format value
FEA        ends
```

Note: The length of the name does not include the trailing null. Names are not limited to 8 characters, but may be up to 255 characters long.

BUFLen is the size of the data buffer.

BUFFER is the data buffer used to pass information.

InfoLevel is the level of information required.

Returns: Successful Completion if carry is not set.

Unsuccessful Completion if carry is set. AX = any valid extended error. If the error was caused by attempting to set an invalid valid extended attribute then offError in the EAOP structure contains the offset of the offending attribute.

Remarks On a get extended attributes call the results are the same as the protect mode API call DOSQFileInfo when BX<>0FFFFH or DOSQPathInfo when BX=0FFFFH.

On a set extended attributes call the results are the same as the protect mode API call DOSSetFileInfo when BX<>0FFFFH or DOSSetPathInfo when BX=0FFFFH.

The FAPI support for utilities calls for get and set information have the same info levels and input/output buffer structures as the protect mode API call DOSQFileInfo/DOSSetFileInfo when BX<>0FFFFH or DOSQPathInfo/DOSSetPathInfo when BX=0FFFFH.

Note: See DOSQFileInfo, DOSQPathInfo, DOSSetFileInfo, DOSSetPathInfo for more information.

+ 2.1.0.1.2 ENUMERATE EXTENDED ATTRIBUTES (INT 21H, 6EH):

+ Purpose A new Interrupt 21H will be created to allow the enumeration of Extended Attributes. This new function is modelled after the OS/2 DOSENUMATTRIBUTE function. It combines the functions currently available with DOSENUMATTRIBUTE and DOSFCTL function 2.

+ Format Calling Sequence:

```
MOV AH,6EH      ; Enumerate EAs
MOV AL,0        ; Reserved
LDS SI,ParamPkt ; DS:SI = Pointer to buffer
INT 21H
JC ERROR
```

Where: ParamPkt is a structure containing the parameters to pass to the call. The structure is defined as follows, and is derived from the argument frame for DOSENUMATTRIBUTE:


```

+ EnumEAStruc      struc      ; Enumerate EAs parm structure
+   eeas_Reserved   dd         0      ; Reserved. Must be zero
+   eeas_InfoLevel  dd         ?      ; Level of information requested
+   eeas_EnumCnt    dd         ?      ; Count of EAs
+   eeas_EnumBufSize dd         ?      ; Size of return buffer
+   eeas_EnumBuf     dd         ?      ; Pointer to return buffer
+   eeas_EntryNum   dd         ?      ; Entry in EA from which to start
+   eeas_FileRef    dd         ?      ; Pointer to Handle or Path
+   eeas_RefType    dw         ?      ; Type of reference in FileRef
+ EnumEAStruc      ends

```

eeas_Reserved is reserved and must be set to zero.

eeas_InfoLevel contains the level of information desired. Level 0x00000001 information is returned in the following format:

```

+ struct {
+   unsigned char reserved; /* must be 0
+   unsigned char cbName;   /* length of name excluding NULL
+   unsigned short cbValue; /* length of value
+   unsigned char szName[]; /* asciiz name
+ };

```

The fields in the structure are on a one-to-one mapping for the fields in an FEA structure.

The information for the next EA will be stored adjacent to the previous one. An application that wishes to continue on from one DosEnumAttribute call would have to set EntryNum to the previous value plus the returned value in EnumCnt.

The size of buffer needed to hold an EA can be obtained from the enumerated information. An EA would occupy:

```

+ one byte (for fea_Reserved) +
+ one byte (for fea_cbName) +
+ two bytes (for fea_cbValue) +
+ value of cbName (for the name of the EA) +
+ one byte (for terminating NULL in fea_cbName) +
+ value of cbValue (for the value of the EA)

```

eeas_EnumCnt contains, on input, the number of EAs whose information has been requested. The system will change

this to the number actually returned. An EnumCnt of -1 is treated specially in that it will return as many entries as will fit in the buffer specified.

eeas_EnumBufSize is the size of the buffer at eeas_EnumBuf.

eeas_EnumBuf is a pointer to the buffer where the requested information is returned.

eeas_EntryNum is the ordinal of the entry in the EA list from which to start copying information into EnumBuf. The EntryNum value of zero is reserved, and the first EA is indicated by an EntryNum of 1.

eeas_FileRef contains a pointer to a WORD handle obtained from Open/Create, or to an ASCIIZ name of a file or directory, as indicated by eeas_RefType.

eeas_RefType indicates whether eeas_FileRef points to a handle or an ASCIIZ name. RefType = 0 indicates a handle; RefType = 1 indicates that FileRef points to an ASCIIZ name of a file or directory.

```

+ /*
+   /*/Unsuccessful Completion if carry is set. AX = any valid extended
+   /*/error.

```

+ Remarks The size of EnumBuf indicated in EnumBufSize should be at least big enough to hold the information for one EA. This will depend on the length of the EA's name.

+ It is important to note that the use of EntryNum in no way guarantees the specific ordering of the EAs. It is only a tool to enumerate the EAs, and should not be used to "back up to the nth EA", since the EAs are subject to change by other tasks. So, for example, the EA returned when EntryNum is 11 may not necessarily be the EA returned when EntryNum is 11 for a subsequent call if another task had performed a SET operation on the EAList.

+ No explicit EA sharing is done for DosEnumAttribute. Implicit sharing exists if the caller passes in the handle of an open file, since sharing access to the associated file is required to modify its EAs. No sharing is done if the caller passes in the pathname. This means that if some other process is editing the EAs, and changes them between two calls to DosEnumAttribute, inconsistent results may be returned (i.e. see same value twice, miss seeing values, etc.) To prevent the modification of EAs for the handle case, make sure that the file is open in deny-write mode. To prevent the modification of EAs for the pathname case, open the file in deny-write mode.

2.1.0.1.3 QUERY FILE SYSTEM FOR MAX EA SIZE (INT 21H, 6FH):

Purpose A new Interrupt 21H will be created to allow querying of a file system for the maximum sizes of Extended Attributes supported.

Format Calling Sequence:

```
MOV AH,6FH      ; Query file system about EA sizes
MOV AL,0        ; Reserved
LDS SI,EASizeBuf ; DS:SI = Pointer to buffer
LES DI,PathName ; ES:DI = Pointer to path name
INT 21H
JC ERROR
```

Where: EASizeBuf is a structure that will hold the returned sizes from the file system. The structure is defined as follows:

```
EASizeBufStruc (
    unsigned short easb_MaxEASize /* Max. size of an
                                   individual EA supported */
    unsigned long easb_MaxEAListSize /* Max. Full EA list size
                                     supported */
)
```

PathName is an ASCIIZ string that contains a path that is to be used to identify the file system that is to be queried. The file system that will be queried will be the one that is currently mounted on the volume to which PathName refers.

Unsuccessful Completion if carry is set. AX = any valid extended error.

Remarks The information returned can be used by an application to decide whether an EA operation will succeed on a given file system before the operation is attempted. This will have the effect of improved performance on certain environments, e.g. across a network.

2.1.0.1.4 EXTENDED OPEN/CREATE (INT 21H, 6CH):

Purpose A new Real Mode Box OPEN Interrupt 21H will be created to allow Extended Attributes to be passed. This new function is modelled after the OS/2 OPEN2 function. It combines the functions currently available with OPEN, CREATE and CREATE New. When

creating, any Extended Attribute is allowed.

Format Calling Sequence:

```
MOV AH,6CH      ; Extended open2
MOV AL,0        ; for non Extended Attribute (ES:DI NOT
                                   required)

OR

MOV AL,1        ; for Extended Attribute (ES:DI required)
MOV BX,MODE     ; Open mode
MOV CX,ATTR     ; Create attribute (ignored if file
                                   already exists)
MOV DX,FLAG     ; Function control
LDS SI,FILE_NAME ; Name to open or create
LES DI,EABuf    ; Ext attr EAOP structure
INT 21H
JC ERROR
```

Where: MODE

FORMAT : 0WFCC0000ISSSOAAA

AAA=Access code 0=Read
1=Write
2=Read/Write

SSS=Sharing mode 0=Compatibility
1=Deny Read/Write
2=Deny Write
3=Deny Read
4=Deny None

C Cache/No-Cache

The file is opened as follows:

If C = 0; I/O to the file need not be done through the disk driver cache.

If C = 1; It is advisable for the disk driver to cache the data in I/O operations on this file.

This bit is an ADVISORY bit, and is used to advise FSDs and device drivers on whether it is worth caching the

+ data or not. This bit, like the write-through bit, is a #
+ per-handle bit. It is not inherited by child processes. #

I 0=Pass handle to child, 1=No inherit

F 0=INT 24H, 1=Return error #
On this open and any I/O to this handle

W 0=No commit, 1=Auto-commit on write

ATTR is defined as follows. Note that the LIST of
attributes will override ATTR's settings.

Bit #	Meaning
15	reserved = 0
14 to 9	reserved = 0
8	reserved = 0
7	reserved = 0
6	reserved = 0
5	Archive
4	Directory
3	Volume
2	System
1	Hidden
0	ReadOnly

FLAG

Format=00000000NNNNEEEE

NNNN=Does not exist action
0=Fail, 1=Create

EEEE=Exists action
0=Fail, 1=Open, 2=Replace/Open

EABuf

EABuf EAOP <>

EAOP is a structure used to pass extended attribute
information. See Get/Set Extended Attributes for a full
description of the EAOP structure.

If the caller does not want to pass any extended
attributes then DI must be set to 0FFFFh.

Returns: Successful Completion if carry is not set.

AX = Handle
CX = Action taken code
01H = FILE OPENED
02H = FILE CREATED/OPENED
03H = FILE REPLACED/OPENED

Unsuccessful Completion if carry is set. AX = any valid extended error.

Remarks Any re-creation of a file (ie, Create of file that already exists) does not delete any Extended Attributes defined for that file.

2.1.0.1.5 GET/SET MEDIA ID (INT 21H, 69H):

Purpose This function allows the utilities and real mode command shell to query the volume serial number and volume label in real mode.

Since IFS volume labels can be up to 255 characters long (TUGBOAT media call structure only allows for 11 and truncates longer names) there will be a FAPI interface giving the function of DOSQFsInfo and DOSSetFsInfo. This will allow OS/2 utilities to gain access to the full volume label.

The INT 21H, AH=69H API will be unpublished in both OS/2 and TUGBOAT.

Format Calling Sequence:

Get/Set Media ID

```
MOV AH,69H      ; GET/SET Media ID
MOV AL,SUBFUNCTION ; 0 = Get Media ID
                  ; 1 = Set Media ID
MOV BL,DRIVE     ; Drive number
MOV BH,0         ; Reserved
LDS DX,BUFFER    ; Buffer containing information
INT 21H
JC ERROR
```

FAPI support for utilities

```
MOV AH,69H      ; GET/SET Media ID
MOV AL,SUBFUNCTION ; 0 = Get FSInfo
                  ; 1 = Set FSInfo
MOV BL,DRIVE     ; Drive number
MOV BH,Infolevel ; Information level
MOV CX,BUFLEN    ; Buffer size
LDS DX,BUFFER    ; Buffer containing information
INT 21H
JC ERROR
```

Where: SUBFUNCTION is either 0 to query or 1 to set.

DRIVE is the number of the drive to be queried or set.

BUFLen is the size of the data buffer.

BUFFER is the data buffer where information is passed. For the Get/Set Media ID calls the buffer has the following format.

```
MEDIAINFO struc      ; Media ID structure
level dw 0            ; reserved
serial dd ?           ; serial number
volume db 11 dup(' ') ; Volume label
fstype db 8 dup(' ')  ; File system type
MEDIAINFO ends
```

The FAPI support for utilities calls for get and set File system information have the same info levels and input/output buffer structures as the protect mode API calls DOSQFsInfo and DOSSetFsInfo.

Note: See DOSQFsInfo and DOSSetFsInfo for more information.

Returns: Successful Completion if carry is not set, buffer contains the information or information was set from buffer.

Unsuccessful Completion if carry is set.

Remarks The Get Media ID call will return blanks for the file system type in all cases Likewise the Set Media ID call will not set the serial number or file system type.

The INT 21H, AH=69H function will not be published in OS/2 or TUGBOAT.

2.1.0.2 Miscellaneous Enhancements

2.1.0.2.1 QUERY DOS VALUE (INT 21H, 3305H):

Purpose This function allows a Real Mode Box application to query the boot device.

This function is being introduced for OS/2 1.2 compatibility.

Format Calling Sequence:

```
MOV    AX,3305H      ; QUERY DOS VALUE
INT     21H
JC      ERROR
```

Where: No input parameters.

Returns: Successful Completion if DL = boot drive where 1 = A, 2 = B, etc.

Unsuccessful Completion if AL = 0FFh.

Remarks This function is being introduced for OS/2 1.2 compatibility.

2.1.1 Interprocess Communication

To provide the means for allowing processes to communicate with one another, OS/2 supports four general methods of interprocess communication (IPC): flags, pipes, semaphores, and shared memory. All methods except flags work between the threads of one process, as well as inter-process.

The IPC functions provided allow:

- * communication via flags,
- * communication via messages,
- * coordinating execution among several processes,
- * one process to directly control execution of other processes,

2.1.1.1 Communication via flags

Communication via flags is used to allow one process to set an external event flag to another process. The target process must use DosSetSigHandler "-----" on page --- to inform OS/2 that it wishes to intercept any of 3 flags; then, another process may issue a DosFlagProcess indicating which of the flags to signal. The target process will receive control at the signal handler it has defined for that signal.

2.1.1.2 Communication via messages

There are two facilities provided for interprocess communication via messages - pipes and queues.

2.1.1.2.1 Pipes:

For communicating via pipes, the standard OS/2 read and write functions are used. Pipe support is provided only for applications in which the pipe participants are a closely related group of processes. The functions provided are in the DosMakePipe function.

Pipes are a technique by which two related processes may communicate as if

they were doing file I/O. In fact, a program which inherits a pipe handle can not distinguish if its I/O requests to that handle are to a file or pipe.

The storage required, or available, for a pipe I/O request to be performed may be a consideration. Pipes are effectively fixed length in nature with the maximum that any pipe can hold being 64 kb at any one time. If a pipe is full, further write requests will block until some data is removed from the pipe.

2.1.1.2.2 Queues:

The following functions are provided for communicating via Queues:

DosCloseQueue Close a Queue.

DosCreateQueue Create a Queue.

DosOpenQueue Open a Queue.

DosPeekQueue Get an element from Queue, but do not remove it.

DosPurgeQueue Purge all entries from a Queue.

DosQueryQueue Find how many elements are in Queue.

DosReadQueue Get an element from a Queue and remove it.

DosWriteQueue Adds an element to a Queue.

2.1.1.2.3 Comparing Pipes and Queues:

Pipes are a technique by which two processes may communicate as if they were doing file I/O. In fact, a program which lists a file may have its input or output specified as being a pipe and the program would operate without change. The pipe support would insure that all data read or written by the program went to a pipe rather than to a file or a printer.

Programs which use queues must be designed and coded with the concepts and system calls defined by queuing in mind. These calls are not at all like the file I/O calls and the processing of the data is completely different from that used with pipes.

Another characteristic of interest is the relative performance offered by pipes or queues. Queues are more efficient than pipes as only a pointer to the data is passed.

The storage required, or available, for the IPC to be performed is often a consideration. Pipes are effectively fixed length in nature (not over 64 kb

in length) while queues may be of relatively unbounded length since queue messages need not be contained in one segment.

While a pipe may contain as many messages as will fit in the pipe's data segment, the total number of messages which may be placed into a single queue is about 4000.

The application designer must take into account all these factors balancing the performance requirements against the storage usage characteristics of each solution.

2.1.1.3 Coordinating execution among several threads

For coordinating the execution of several threads, OS/2 provides several functions. In a multitasking application environment, these functions allow the various processes to have a great deal of control over one another's execution. The functions provided include:

- # * Semaphores - RAM based and file system based,
- # * Starting and stopping a thread's execution,

2.1.1.3.1 Semaphores:

Two types of semaphores are provided in OS/2:

RAM Semaphores which are defined by the requesting program allocating a double-word of storage and using the address in the semaphore calls provided below. RAM semaphores are a minimal function mechanism with OS/2 performing no resource management services such as freeing when the owner terminates.

System Semaphores which are defined by OS/2 in response to a create semaphore system call. Once created, they may be accessed by separate processes and OS/2 will provide full resource management including freeing and notification when the owner terminates.

The functions provided for controlling access to a serially reusable resource (via either RAM or System semaphores) are:

DosSemClear Clear a semaphore.

DosSemRequest Obtain a semaphore.

To allow a simple wait/post type of signalling between threads, several
functions are provided:

DosSemSet Set a semaphore.

DosSemSetWait Set a semaphore and wait for it to be cleared.

DosSemWait Wait for a semaphore to be cleared.

DosMuxSemWait Wait for any one of many semaphores to be cleared.

And, these are provided to have OS/2 allocate, provide access to, and delete
system semaphores:

DosSemClose Close a system semaphore.

DosSemCreate Create a system semaphore.

DosSemOpen Open a system semaphore.

2.1.1.3.2 Starting and Stopping a Thread's Execution:

For explicitly controlling when a thread may execute, the following
functions are provided:

DosResumeThread Restart a thread's execution.

DosSuspendThread Suspend a thread's execution.

The details of each of these categories will now be explored:

2.1.1.3.3 DosFlagProcess - Set a Process's External Event Flag:

Purpose DosFlagProcess allows one process to set an "external event" flag
on another. The target process can detect the triggering of this
flag via DosSetSigHandler. By default, the event flag is ignored.

Format Calling Sequence:

EXTRN DosFlagProcess:FAR

PUSH WORD ProcessID ; Process ID to flag

PUSH WORD Action ; Indicate to send to process or subtree
PUSH WORD Flagnum ; Flag number
PUSH WORD Flagarg ; Flag argument
CALL DosFlagProcess

where ProcessID is the process ID of the process to which the
flag is to be sent.

Action indicates to flag only the specified process or
all descendents of the process.

* If value=0 is coded, all descendents will be
notified.

* If value=1 is coded, only the indicated process will
be notified.

Flagnum is the number of the flag to set:

0 - flag A

1 - flag B

2 - flag C

FlagArg is an argument to be passed to the indicated
process(es).

Returns: IF AX = 0

NO error

ELSE

AX = Error Code:

* Invalid Action

* Invalid Process ID

* Process refuses flag

* Signal already pending

* Process is zombie

Remarks. DosFlagProcess does not actually set a static flag which can be
checked at a later time; it causes a "flag event" to occur for the
specified process(es). By default, a flag event is ignored by a
process. A process may use DosSetSigHandler "-----"
"-----" on page --- to be alerted, via the signal mechanism,
when such a flag event occurs. A process may also specify that

the flag action is to be ignored and that an error code is to be returned to the flagger. The "Process is zombie" error code indicates that the flagged process has died but its' parent has not yet done a DOSCWAIT to get its' return code.

2.1.1.4 Pipes

Pipes are an IPC mechanism based on the file I/O concept.

When comparing pipes with files:

* similarities

- data is communicated to pipes by the standard DosRead and DosWrite system calls,
- pipes are closed by the standard DosClose system call.

* differences

- pipes are created by DosMakePipe rather than one of the file create requests,
- pipes need not be OPENed before being accessed, only the DosMakePipe is necessary,
- pipes are implemented via an in storage buffer mechanism rather than having their data maintained on disk,
- when writing to a file, the requesting thread will be blocked only while doing file I/O. When writing to a pipe, the requesting thread will block if the pipe reader allows the pipe to fill up.
- writes to a pipe will not be interspersed.

A thread issuing a write to a pipe will be blocked until it can write all of the data specified to the pipe. No other thread may write to that pipe until the write in progress completes. Any thread that attempts to write to the pipe is blocked.

- reading data from a pipe removes that data from the pipe, subsequent reads will not find that data.

Pipes are inherited the same as files. A using process would typically create a pipe, then start a child process (who would inherit the pipe handles) and communicate to the child process with the pipe's handles.

2.1.1.4.1 DosMakePipe - Create a Pipe:

Purpose DosMakePipe creates a Pipe.

Format Calling Sequence:

EXTRN DosMakePipe:FAR

PUSH@ WORD ReadHandle ; Address to place the read handle
PUSH@ WORD WriteHandle ; Address to place the write handle
PUSH WORD PipeSize ; Size to reserve for the pipe
CALL DosMakePipe

where ReadHandle is the address of a word where the read handle for the pipe is to be placed.

WriteHandle is the address of a word where the write handle for the pipe is to be placed.

PipeSize is the size, in bytes, of storage to reserve for this pipe.

Returns: IF AX = 0

NO error

ELSE

AX = Error Code:

* Not enough memory, pipe can not be created

Remarks Pipes are provided for use within a closely related group of processes. There are no control or permission mechanisms or checks performed on operations to pipes.

Whenever there is not enough space in a pipe for the data being written, the requesting thread will be blocked until enough data has been removed to allow its write request to be satisfied. If the size parameter is zero, then the pipe will be created with the default size (this size is unknown at this time but will likely be in the range of 1 to 8 kb.)

A pipe will be deleted when all users close the handles. If there are two process communicating by a pipe and the process reading the pipe ends, the next DosWrite to that pipe will return the "write to a broken pipe" return code.

The value in PipeSize is an advisory size in that that may not be the actual amount of memory allocated by the system for the pipe buffer. If a size of zero is specified, then the default size is used for the buffer.

2.1.2 Named Pipes

Named pipes provide I/O between arbitrary (unrelated) processes. Anonymous pipes provide I/O only between a parent and descendent (related) process.

Without named pipe I/O, the only way two unrelated processes in OS/2 can communicate is via shared memory, or via queues which depend on passing references to shared memory. This means that inter-process communication necessitates undesirable coupling between unrelated processes, and such coupling may be impractical to achieve if the processes are supplied by independent vendors (e.g., a database service process and a user interface process).

Anonymous pipes, unlike other file system resources, cannot be virtualized across a network. Named pipes can be virtualized exactly like files (i.e., it is possible to write a network requester for named pipes). Allowing pipe I/O to work across a network the same way that file I/O can is both desirable for architectural consistency and valuable for application development.

These are the error codes specific to Named Pipes API. These calls may also return standard OS/2 error codes.

- # * ERROR_BAD_PIPE (230) - Non-existent pipe or bad operation.
- # * ERROR_PIPE_BUSY (231) - Pipe is busy.
- # * ERROR_BROKEN_PIPE (232) - Other end of pipe is closed.
- # * ERROR_PIPE_NOT_CONNECTED (233) - Pipe was disconnected by server.
- # * ERROR_MORE_DATA (234) - More data is available.

Named Pipe Function Call Summary

DosCallNmPipe Perform a "procedure call" transaction via a named pipe.

DosConnectNmPipe Waits for a new client to open a named pipe.

DosDisconnectNmPipe Forces a named pipe closed.

DosMakeNmPipe Create a named pipe.

DosPeekNmPipe Read named pipe contents without removal.

DosQNmPipeHandState Query named pipe specific handle state information

DosQNmPipeInfo Query named pipe information

DosRawReadNmPipe Read a named pipe directly.

DosRawWriteNmPipe Write to a named pipe directly.

DosSetNmPFHandInfo Set named pipe specific handle state information

DosTransactNmPipe Perform a transaction to a message pipe.

DosWaitNmPipe Waits for an available named pipe instance.

2.1.2.1 Named Pipe Function Calls

2.1.2.1.1 DosCallNmPipe:- Perform a procedure call via a message pipe.:

Purpose Provide a "procedure call" transaction via a message pipe.

Format Calling Sequence:

```
EXTRN DosCallNmPipe:FAR
```

```
PUSH@ ASCIIZ FileName      ; Pipe name
PUSH@ OTHER InBuffer       ; Write buffer address
PUSH WORD InBufferLen      ; Write buffer length
PUSH@ OTHER OutBuffer      ; Read buffer address
PUSH WORD OutBufferLen     ; Read buffer length
PUSH@ WORD BytesOut        ; Bytes read (returned)
PUSH DWORD TimeOut         ; Maximum wait time
CALL DosCallNmPipe
```

where FileName is the ASCIIZ name of the pipe to be opened.
Pipes are named \PIPE\FileName

InBuffer is address of buffer to write to the pipe.

InBufferLen is the number of bytes to be written.

OutBuffer is address of buffer for returned data.

OutBufferLen is the maximum size (number of bytes) of
returned data.

BytesOut is where the system returns the number of bytes
actually read (returned).

TimeOut is the maximum time to wait for pipe
availability.

Returns: IF ERROR (AX not = 0)

AX = Error Code:

- * ERROR_FILE_NOT_FOUND
- * ERROR_BAD_PIPE
- * ERROR_PIPE_BUSY
- * ERROR_PIPE_NOT_CONNECTED
- * ERROR_MORE_DATA

Remarks This routine has the combined effect on a named pipe of DosOpen,
DosTransactNmPipe, DosClose. It provides a very efficient means
of implementing local and remote procedure-call (RPC) interfaces
between processes.

2.1.2.1.2 DosConnectNmPipe:- Waits for a new client to open a pipe.:

Purpose Enables a new client to obtain handle-based access to a named pipe
through DosOpen.

Format Calling Sequence:

```
EXTRN DosConnectNmPipe:FAR
```

```
PUSH WORD Handle ; Pipe handle
CALL DosConnectNmPipe
```

where Handle is the handle of the named pipe returned by
DosMakeNmPipe.

Returns: IF ERROR (AX not = 0)

AX = Error Code:

- * ERROR_BAD_PIPE
- * ERROR_PIPE_NOT_CONNECTED
- * ERROR_BROKEN_PIPE
- * ERROR_INTERRUPT

Remarks DosConnectNmPipe causes a newly made or disconnected named pipe to enter a listen state that will accept a DosOpen from a client (DosOpen to a pipe not in the listen state will fail). If the client end of a pipe is currently open, DosConnectNmPipe returns immediately and has no effect. If the client end is not open, DosConnectNmPipe either waits until it is open (if blocking mode is set) or else returns immediately with error PIPE_NOT_CONNECTED (if non-blocking mode is set). If the pipe was previously opened but has been closed by the client and not yet disconnected by the server, DosConnectNmPipe always returns ERROR_BROKEN_PIPE. Multiple DosConnectNmPipe can be issued in non-blocking mode; the first one puts the pipe into listen state (if it is not already open or closing) and subsequent ones simply test the pipe state. If DosConnectNmPipe is called by the requestor (client) end of the pipe, the error BAD_PIPE is returned. If the wait (in blocking mode only) for the client open was interrupted, the error INTERRUPT is returned.

2.1.2.1.3 DosDisconnectNmPipe:- Forces a named pipe closed.:

Purpose Forces a named pipe to close.

Format Calling Sequence:

```
EXTRN DosDisconnectNmPipe:FAR
```

```
PUSH WORD Handle ; Pipe handle
CALL DosDisconnectNmPipe
```

where Handle is the handle of the named pipe returned by DosMakeNmPipe.

Returns: IF ERROR (AX not = 0)

AX = Error Code:

* ERROR_BAD_PIPE

Remarks If the client end of the pipe is currently opened, DosDisconnectNmPipe forces it closed (the client gets an error code on its next operation). Note that this may discard data which has not yet been read by the client. If the client end is

currently closing (DosClose has been issued), DosDisconnectNmPipe acknowledges the close and makes the pipe available to be reopened. A client that gets forced off a pipe by a DosDisconnectNmPipe must still issue DosClose to close its end of the pipe.

2.1.2.1.4 DosMakeNmPipe - Create a named pipe.:

Purpose Creates the specified named pipe and returns its handle.

Format Calling Sequence:

```
EXTRN DosMakeNmPipe:FAR
```

```
PUSH@ ASCIIZ FileName ; Pipe name
PUSH@ WORD PipeHandle ; Returned pipe handle
PUSH WORD OpenMode ; DOS open mode of pipe
PUSH WORD PipeMode ; Pipe open mode
PUSH WORD OutBufSize ; Advisory outgoing buffer size
PUSH WORD InBufSize ; Advisory incoming buffer size
PUSH DWORD Timeout ; Timeout for DosWaitNmPipe
CALL DosMakeNmPipe
```

where FileName is the ASCIIZ name of the pipe to be opened. Pipes are named \PIPE\FileName

PipeHandle is where the system returns the handle of the named pipe that is created.

OpenMode is the Dos OpenMode of the named pipe and consists of the following bit fields. They are the:

* Inheritance flag

* Write-through flag

* Access field

The bit field mapping is shown as follows:

```
Open Mode  5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
bits      0 W * * * * * I 0 0 0 * A A A
```

W File Write-through

The file is opened as follows:

If W = 0; Write-behind to remote pipes is allowed.
If W = 1; Write-behind to remote pipes is not allowed.

Write behind only has meaning in the case of remote pipe. OS/2 will in certain cases locally buffer data that is written to a pipe, and not send it across the net to the remote end until a future time. The Write through bit lets the application prevent this behavior, so that data will be sent across the net as soon as it is written by the application.

I Inheritance Flag

If I = 0; Pipe handle is inherited by a spawned process resulting from a DosExecPgm call.
If I = 1; Pipe handle is private to the current process and cannot be inherited.

This bit is not inherited by child processes.

AAA Access Mode

The pipe access is assigned as follows:

If A = 000; In-bound pipe (client to server)
= 001; Out-bound pipe (server to client)
= 010; Full duplex pipe (server to/from client)

Any other value is invalid.

* Bit is not applicable.

PipeMode defines the pipe-specific mode parameters and consists of the following bit fields. They are the:

- * Blocking
- * Type of named pipe
- * Read mode
- * Instance count

The bit field mapping is shown as follows:

Pipe Mode 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
bits B * * * T T R R]----ICount---

B Blocking:

The pipe is defined as follows:

If B = 0; Reads/Writes block if no data available.
If B = 1; Reads/Writes return immediately if no data available.

Reads normally block until at least partial data can be returned. Writes by default block until all bytes requested have been written. Non-blocking mode (B=1) changes this behavior as follows:

1. DosRead will return immediately with BytesRead=0 if no data is available.
2. DosWrite will return immediately with BytesWritten=0 if no data is available. Otherwise, the entire data area will be transferred.

TT Type of named pipe.

If TT = 00; Pipe is a byte stream pipe.
= 01; Pipe is a message stream pipe.

All writes to message stream pipes record the length of the write along with the written data (see DosWrite). The first two bytes of each message represents the length of that message and is called the message header. A header of all zero's is reserved. Zero length messages are not allowed (because OS/2 no-ops zero-length I/Os).

RR Read Mode

If RR = 00; Read pipe as a byte stream.
= 01; Read pipe as a message stream.

Message pipes can be read as byte or message streams, depending on the setting of RR. Byte pipes can only be read as byte streams (see DosRead).

ICount Byte wide (8-bit) count to control pipe instancing. Referred to as Instance Count. When making the first instance of a named pipe, ICount specifies how many instances can be created:

```
#
#      * 1 ==> this can be the only instance (pipe is
#      unique),
#
#      * -1 ==> the number of instances is unlimited,
#
#      * 0 ==> reserved value.
```

```
#
#      Subsequent attempts to make a pipe will fail if
#      the maximum number of allowed instances already
#      exists. The ICount parameter is ignored when
#      making other the first instance of a pipe. When
#      multiple instances are allowed, multiple clients
#      can simultaneously DosOpen to the same pipe
#      name and get handles to distinct pipe instances.
```

```
#
#      OutBufSize is an advisory to the system of the number of bytes to
#      allocate for the out-going buffer.
```

```
#
#      InBufSize is an advisory to the system of the number of bytes to
#      allocate for the incoming buffer.
```

```
#
#      Timeout is the default value for the timeout parameter to
#      DosWaitNmPipe. This value may be set only at the creation of the
#      first instance of the pipe name. If at that time the value is
#      zero, a system wide default value (50 ms) will be chosen.
```

```
# Returns: IF ERROR (AX not = 0)
```

```
#      AX = Error Code:
```

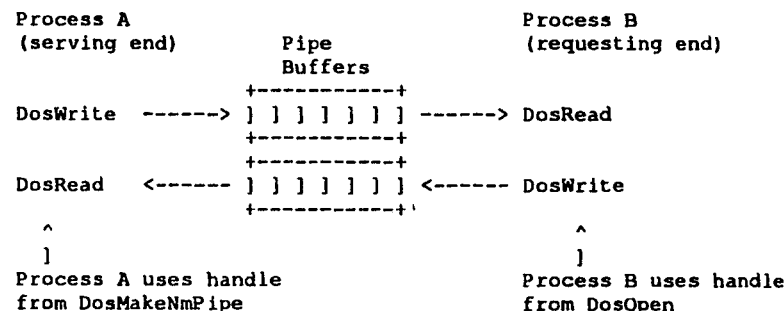
```
#      * ERROR_PATH_NOT_FOUND
#
#      * ERROR_INVALID_PARAMETER
#
#      * ERROR_PIPE_BUSY
#
#      * ERROR_OUT_OF_STRUCTURES
#
#      * ERROR_NOT_ENOUGH_MEMORY
```

```
# Remarks How the full duplex pipe works:
```

```
#
#      The concept is that one process uses DosMakeNmPipe to create a
#      named pipe, and then waits for another process to open the pipe by
#      name via DosOpen. The pipe maker is called the "serving" end of
#      the pipe. The pipe opener (client) is called the "requesting"
#      end.
```

```
#
#      Once a pipe has been made and successfully opened, each end has a
#      handle associated with it. The serving end gets its handle from
```

DosMakeNmPipe. The requesting end gets its handle from DosOpen. Unlike for anonymous pipes, each end can both read and write its one handle. Anything that one end writes the other end will be able to read and vice versa. This is because there are in fact two pipe buffers for a named pipe, one for each direction of flow.



Here is an example to show how a named pipe is created, opened, and read/written by each end. Comments follow the example that explain the role of the DosConnectNmPipe/DosDisconnectNmPipe calls.

Process A (serving end)	Process B (requesting end)
/* First make a pipe */ DosMakeNmPipe("\\pipe\\foo",handle1)	
]	
v	
/* Wait for requester to open it */ DosConnectNmPipe(handle1)	
:	
:	/* Open pipe */ DosOpen("\\pipe\\foo",handle2)
<-----	
]	
v	
/* Read client request */ DosRead(handle1,buffer1)	
:	
<-----	/* Write request to pipe */ DosWrite(handle2,buffer2)
]	
]	
/* Process client's request */	/* Read response from pipe */ DosRead(handle2,buffer2)
:	
:	
v	:

```

/* Write response */
DosWrite(handle1,buffer1) ----->
                                v
                                ]
                                ]
                                ]
                                v
/* Prepare for next client */
DosDisconnectNmPipe(handle1)
                                v
/* Wait for another client */
DosConnectNmPipe(handle1)
:

```

It is important to understand the role of the DosConnectNmPipe/DosDisconnectNmPipe calls versus DosOpen/DosClose.

A serving process creates a named pipe and then must wait for a client to open it. The DosConnectNmPipe accomplishes this. It takes the handle for the pipe and waits for an open to happen (or returns status indicating that the pipe is already open). Until some client opens the pipe via DosOpen, it is in the "listening" state.

Once a pipe is open, the serving and requesting ends have a conversation using DosRead and DosWrite (or DosTransactNmPipe, which performs DosWrite followed by DosRead on the pipe). The pipe acts like a full duplex channel between the two processes.

At some point, the requesting end may choose to close its end of the pipe, in which case the serving end will eventually get:

1. An EOF on a future read that finds the pipe empty or
2. The error BROKEN_PIPE when it tries to do a write to the pipe.

The serving end must "acknowledge" the client's close of the pipe by doing a DosDisconnectNmPipe. The serving end's handle will now be invalid for I/O until another DosConnectNmPipe is done to await the next DosOpen. Until a close has been acknowledged by DosDisconnectNmPipe, repeated reads of the pipe by the serving end will continue to return EOF, and other requesters that attempt to open the pipe will get error PIPE_BUSY.

These rules make sure that the serving end has a well controlled way to know when the requesting end of the pipe has been closed, and to let the serving end control when it is acceptable for another requester to open the pipe.

The serving end can destroy a named pipe at any time. It uses the following OS/2 call sequence to do this:

```

* DosDisconnectNmPipe (handle1)
* DosClose (handle1)

```

DosDisconnectNmPipe will dissociate the requesting end's file handle from the named pipe data structures and allow DosClose to terminate the last reference to the pipe.

The serving end's DosClose frees its pipe handle, however, the pipe buffer will not be released until the requesting end has also issued a DosClose on its handle to the pipe. In this case, the requester will get the error PIPE_NOT_CONNECTED on its next DosRead or DosWrite, and will have to issue the DosClose.

The serving end may also issue a DosClose to free its pipe handle without a preceding DosDisconnect. The effect of this is that the requestor can read any of the remaining data in the pipe buffer without getting the error PIPE_NOT_CONNECTED on every DosRead operation.

The concepts of disconnecting and closing are separated so that the serving end can create a pipe and reuse it serially for conversations with different requesters. Without this concept, the serving end would have to delete and recreate the pipe for each conversation.

2.1.2.1.5 DosPeekNmPipe:- Peek into a named pipe.:

Purpose Read pipe without removing the read data from the pipe.

Format Calling Sequence:

EXTRN DosPeekNmPipe:FAR

```
PUSH WORD Handle      ; Pipe handle
PUSH@ OTHER Buffer     ; Address of user buffer
PUSH WORD BufferLen    ; Buffer length
PUSH@ WORD BytesRead  ; Bytes read
PUSH@ DWORD BytesAvail ; Bytes available
PUSH@ WORD PipeState  ; Pipe state
CALL DosPeekNmPipe
```

where Handle is the handle of the named pipe returned by DosMakeNmPipe or DosOpen.

Buffer is address of the output buffer.

BufferLen is the number of bytes to be written.

BytesRead is where the system returns the number of bytes actually read.

BytesAvail is a 4-byte buffer where the system returns the number of bytes that were actually available. This buffer is structured as follows:

- * 2 bytes - Bytes left in pipe (including message header bytes)
- * 2 bytes - Bytes left in current message (zero for a byte stream pipe)

PipeState is where the system returns a value representing the state of the named pipe.

- * If value=1, the state of the pipe is Disconnected
- * If value=2, the state of the pipe is Listening
- * If value=3, the state of the pipe is Connected
- * If value=4, the state of the pipe is Closing

Returns: IF ERROR (AX not = 0)

AX = Error Code:

* ERROR_BAD_PIPE

* ERROR_PIPE_NOT_CONNECTED

Remarks DosPeekNmPipe acts like DosRead except as follows:

1. The bytes read are not removed from the pipe.
2. The peek may return only part of a message (that part currently in the pipe), even if the size of the peek would accommodate the whole message.
3. DosPeekNmPipe never blocks, regardless of the blocking mode.
4. Additional information about the status of the pipe and remaining data are returned. The caller can use this, for example, whether the peek returned all of the current message or whether the pipe is at EOF (pipe is at EOF when there are no bytes left in the pipe and Status is Closing or Disconnected).

PipeState is where the system returns the state of the pipe (Disconnected, Listening, Connected, Closing).

The four states of a named pipe are defined as follows:

- * Disconnected - after a DosMakeNmPipe or a DosDisconnectNmPipe.
 - Indicates that the pipe has no current requesting end and that it is not willing to accept an open. The serving end must issue DosConnectNmPipe to put the pipe into the listening state before an open will be accepted. A pipe will be in the disconnected state
 1. Immediately after the DosMakeNmPipe but before the first DosConnectNmPipe or
 2. Immediately after a DosDisconnectNmPipe but before the next DosConnectNmPipe.
- * Listening - after a DosConnectNmPipe.
 - Puts pipe into a state where it will accept a DosOpen from a requester. A pipe not in listening state cannot be opened (DosOpen will return "pipe busy").
- * Connected - after the first DosOpen to the pipe.
 - Indicates that a requester has DosOpened the pipe. The serving and requesting ends are able to do I/O to the pipe

(ie, there are valid handles for both ends).

* Closing - after the last DosClose to the pipe from either the server or requester end.

- The requesting end of the pipe has issued DosClose for all dups of the handle it had to the pipe, i.e., the requesting end of the pipe is closed. The serving end will get EOF on reads to the pipe after it has successfully read all buffered data; on writes, serving end will get error BROKEN_PIPE. The Serving end must issue DosDisconnectNmPipe or DosClose to acknowledge the close of the requesting end. If DosDisconnectNmPipe is used, the pipe can be reused by issuing DosConnectNmPipe, which allows another requester to open it. If DosClose is used, the pipe is deallocated.

2.1.2.1.6 DosQNmPHandState - Query Named Pipe Handle State:

Purpose Return named pipe-specific handle state information.

Format Calling Sequence:

EXTRN DosQNmPHandState:FAR

PUSH WORD Handle ; Pipe handle
PUSH@ WORD PipeHandleState ; Pipe handle state
CALL DosQNmPHandState

where Handle is the handle of the named pipe returned by DosMakeNmPipe or DosOpen.

PipeHandleState is the named pipe handle state and consists of the following bit fields. They are the:

- * Blocking
- * Server or requestor end
- * Type of named pipe
- * Read mode

* Instance count

The bit field mapping is shown as follows:

Pipe Mode 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
bits B E * * T T R R]----ICount----

B Blocking:

The pipe is defined as follows:

If B = 0; Reads/Writes block if no data available.

If B = 1; Reads/Writes return immediately if no data available.

Reads normally block until at least partial data can be returned. Writes by default block until all bytes requested have been written. Non-blocking mode (B=1) changes this behavior as follows:

1. DosRead will return immediately with BytesRead=0 if no data is available.
2. DosWrite will return immediately with BytesWritten=0 if no data is available. Otherwise, the entire data area will be transferred.

E End-point of named pipe.

If E = 0; Handle is the client end of a named pipe
= 1; Handle is the server end of a named pipe

TT Type of named pipe.

If TT = 00; Pipe is a byte stream pipe.
= 01; Pipe is a message stream pipe.

RR Read Mode

If RR = 00; Read pipe as a byte stream.
= 01; Read pipe as a message stream.

ICount Byte wide (8-bit) count to control pipe instancing. Referred to as Instance Count. When making the first instance of a named pipe, ICount specifies how many instances can be created:

- * 1 ==> this can be the only instance (pipe is unique),
- * -1 ==> the number of instances is unlimited,
- * 0 ==> reserved value.

Returns: IF ERROR (AX not = 0)

AX = Error Code:

- * ERROR_BAD_PIPE
- * ERROR_PIPE_NOT_CONNECTED

Remarks At the serving end, the values returned by DosQNmPipeInfo are those originally established by DosMakeNmPipe or a subsequent DosSetNmPipeInfo. For the client end, the values returned are those originally established by DosOpen or a subsequent DosSetNmPipeInfo.

2.1.2.1.7 DosQNmPipeInfo - Query a named pipe's Information:

Purpose Returns information for a named pipe.

Format Calling Sequence:

EXTRN DosQNmPipeInfo:FAR

```
PUSH WORD Handle ; Pipe handle
PUSH WORD InfoLevel ; Pipe data required
PUSH@ OTHER InfoBuf ; Pipe data buffer
PUSH WORD InfoBufSize ; Pipe data buffer size
CALL DosQNmPipeInfo
```

where Handle is the handle of the named pipe returned by DosMakeNmPipe or DosOpen.

InfoLevel is the level of the pipe information required.

Level '1' file information is returned in the following format:

- * 2 bytes - Actual size of buffer for outgoing I/O

- * 2 bytes - Actual size of buffer for ingoing I/O
- * 1 byte - Maximum allowed number of pipe instances
- * 1 byte - Current number of pipe instances
- * 1 byte - Length of pipe name
- * Asciz - Name of pipe (including \\ComputerName if remote)

InfoBuf is the storage area where the system returns the requested level of named pipe information.

InfoBufSize is the length of InfoBuf.

Returns: IF ERROR (AX not = 0)

AX = Error Code:

- * ERROR_BUFFER_OVERFLOW
 - DosQNmPipeInfo returns whatever will fit in InfoBuf
- * ERROR_INVALID_LEVEL
- * ERROR_BAD_PIPE

Remarks

2.1.2.1.8 DosRawReadNmPipe:- Direct (raw) read named pipe:

Purpose Read bytes directly from a named pipe, regardless of whether it is a message or a byte stream pipe.

Format Calling Sequence:

EXTRN DosRawReadNmPipe:FAR

```
PUSH WORD Handle      ; Pipe handle
PUSH@ OTHER Buffer     ; Buffer address
PUSH WORD BufferLen    ; Buffer length
PUSH@ WORD BytesRead  ; Bytes read (returned)
CALL DosRawReadNmPipe
```

where Handle is the handle of the named pipe returned by DosMakeNmPipe or DosOpen.

Buffer is address of buffer to receive the data from the pipe.

BufferLen is the number of bytes to be read.

BytesRead is where the system returns the number of bytes actually read (returned).

Returns: IF ERROR (AX not = 0)

AX = Error Code:

* ERROR_PIPE_NOT_CONNECTED

Remarks

For a byte pipe, this is exactly like DosRead. For a message pipe, this is exactly like DosReading the pipe in byte read mode, except message headers will also be returned in the buffer (note that message headers will always be returned in total--never split at a byte boundary).

This function will not be documented to users.

2.1.2.1.9 DosRawWriteNmPipe:- Direct (raw) write named pipe:

Purpose Put bytes directly into a named pipe, regardless of whether it is a message or a byte stream pipe.

Format Calling Sequence:

EXTRN DosRawWriteNmPipe:FAR

```
PUSH WORD Handle      ; Pipe handle
PUSH@ OTHER Buffer     ; Buffer address
PUSH WORD BufferLen    ; Buffer length
PUSH@ WORD BytesWritten ; Bytes written (returned)
CALL DosRawWriteNmPipe
```

where Handle is the handle of the named pipe returned by DosMakeNmPipe or DosOpen.

Buffer is address of data buffer to write to the pipe.

BufferLen is the number of bytes to be written.

BytesWritten is where the system returns the number of bytes actually written (returned).

Returns: IF ERROR (AX not = 0)

AX = Error Code:

* ERROR_BROKEN_PIPE

* ERROR_PIPE_NOT_CONNECTED

Remarks

The data in Buffer should include message headers if it is a message pipe. This call ignores the blocking/nonblocking state and always acts in a blocking manner--it returns only after all bytes have been written.

This function will not be documented to users.

2.1.2.1.10 DosSetNmPipeHandState - Set Named Pipe Handle State:

Purpose Return named pipe-specific handle state information.

Format Calling Sequence:

```

; EXTRN DosSetNmPipeHandState:FAR
;
; PUSH WORD Handle      ; Pipe handle
; PUSH WORD PipeHandState ; Pipe handle state
; CALL DosSetNmPipeHandState
;
; where Handle is the handle of the named pipe returned by
; DosMakeNmPipe or DosOpen.
;
; PipeHandState is the named pipe handle state and
; consists of the following settable bit fields. They are
; the:
;
; * Blocking
;
; * Read mode
;
; The bit field mapping is shown as follows:
;
; Pipe Mode  5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
; bits      B * * * * * R R 0 0 0 0 0 0 0 0
;
; B      Blocking:
;
; The pipe is defined as follows:
;
; If B = 0; Reads/Writes block if no data
;         available.
; If B = 1; Reads/Writes return immediately
;         if no data available.
;
; Reads normally block until at least partial data
; can be returned. Writes by default block until
; all bytes requested have been written.
; Non-blocking mode (B=1) changes this behavior as
; follows:
;
; 1. DosRead will return immediately with
;    BytesRead=0 if no data is available.
;
; 2. DosWrite will return immediately with
;    BytesWritten=0 if no data is available.
;    Otherwise, the entire data area will be
;    transferred.
;
; RR      Read Mode

```

```

; If RR = 00; Read pipe as a byte stream.
; If RR = 01; Read pipe as a message stream.
;
; Returns: IF ERROR (AX not = 0)
;
; AX = Error Code:
;
; * ERROR_INVALID_PARAMETER
;
; * ERROR_BAD_PIPE
;
; * ERROR_PIPE_NOT_CONNECTED
;
; Remarks Note that only the read mode (byte vs message) and
; blocking/nonblocking mode of a named pipe can be changed.
;
; 2.1.2.1.11 DosTransactNmPipe:- Perform a transaction on a message pipe.:
;
; Purpose Perform a transaction (write followed by a read) on a message
; pipe.
;
; Format Calling Sequence:
;
; EXTRN DosTransactNmPipe:FAR
;
; PUSH WORD Handle      ; Pipe handle
; PUSH# OTHER InBuffer   ; Write buffer address
; PUSH WORD InBufferLen  ; Write buffer length
; PUSH# OTHER OutBuffer  ; Read buffer address
; PUSH WORD OutBufferLen ; Read buffer length
; PUSH# WORD BytesOut    ; Bytes read (returned)
; CALL DosTransactNmPipe
;
; where Handle is the handle of the named pipe returned by
; DosMakeNmPipe or DosOpen.
;
; InBuffer is address of buffer to write to the pipe.
;
; InBufferLen is the number of bytes to be written.
;
; OutBuffer is address of buffer for returned data.

```

Microsoft Confidential
OS/2 1.2 IFS Patent Documentation

OutBufferLen is the maximum size (number of bytes) of returned data.

BytesOut is where the system returns the number of bytes actually read (returned).

Returns: IF ERROR (AX not = 0)

AX = Error Code:

- * ERROR_BAD_PIPE
- * ERROR_PIPE_BUSY
- * ERROR_PIPE_NOT_CONNECTED
- * ERROR_MORE_DATA

Remarks This provides an optimum way to implement transaction-oriented dialogs. DosTransactNmPipe will fail if the pipe currently contains any unread data or is not in message read mode. Otherwise the call will write the entire InBuffer to the pipe and then read a response from the pipe into the OutBuffer. The state of blocking/nonblocking has no affect on this call--DosTransactNmPipe does not return until a message has been read into the OutBuffer. If the OutBuffer is too small to contain the response message, the error MORE_DATA will be returned as described for DosRead.

2.1.2.1.12 DosWaitNmPipe:- Wait for an available named pipe instance.:

Purpose Waits for the availability of a named pipe instance.

Format Calling Sequence:

EXTRN DosWaitNmPipe:FAR

PUSH@ ASCIIZ FileName ; Pipe name
PUSH DWORD Timeout ; Maximum wait time
CALL DosWaitNmPipe

where FileName is the ASCII2 name of the pipe to be opened.

Microsoft Confidential
OS/2 1.2 IFS Patent Documentation

Pipes are named \PIPE\FileName

Timeout is the maximum time (milliseconds) to wait for the named pipe to become available. If a zero value is used, the default value specified at DosMakeNmPipe time will be used. If a '-1' value is used, an indefinite wait will be entered.

Returns: IF ERROR (AX not = 0)

AX = Error Code:

- * ERROR_FILE_NOT_FOUND
- * ERROR_PIPE_BUSY
- * ERROR_INTERRUPT

Remarks DosWaitNmPipe allows an application to wait for a server on a pipe for which all instances are currently busy. This call should be used only when ERROR_PIPE_BUSY is returned from a DosOpen call. The call will wait up to Timeout milliseconds (or a default time if Timeout is zero) for a pipe of the name given to become available. When a pipe instance becomes available, it will be given to the highest priority waiting process. Waiting processes of equal priority will be given an available pipe instance based on longest time spent waiting for the named pipe.

2.1.3 I/O Services

OS/2 (TM) provides access to the major character and block devices through system calls. Some devices may be accessed through system calls specific to the device, such as the KBD and VIO calls for the keyboard and display, respectively. A device such as the disk is accessed via file system API. In addition, the file system API may be used to access any named character device, such as LPT1 or COM1.

2.1.4 ANSI Support for Video and Keyboard

The ANSI commands (ESC[...h) to set mode must set, in both OS/2 and DOS modes, the video into the new video modes of the EGA and IBM PS/2 (TM) models 50, 60 and 80 for these codes:

- * 14 is 640x200 color
- * 15 is 640x350 mono
- * 16 is 640x350 color
- * 17 is 640x480 color
- * 18 is 640x480 color
- * 19 is 320x200 color

For enhanced keyboards, Extended Keys (0E0H low bytes) should be supported as distinct keys. PC-DOS will provide this with a /X switch for DEVICE=ANSI.SYS. If /X is omitted, Extended Key values are supported as normal keys (000H low bytes). OS/2 should provide this support in OS/2 and DOS modes and consider PC-DOS compatibility.

2.1.4.1 Device and File Handles

Many system calls use a parameter called a handle. A handle is a 16-bit value which is used to refer to a particular device or file.

2.1.4.2 Handling of I/O

A user process may perform I/O to a character device in one of two modes: raw and cooked. These modes may be set by the user process through the IOCTL facility. The device drivers provided with OS/2 that support raw and cooked mode are:

* Keyboard

In raw mode, data is transferred exactly as it appears and for the length that the user requested. In cooked mode, data may be edited, buffered, and/or translated by the OS/2. The operations that the OS/2 performs for cooked mode I/O are as listed below.

* For a read in cooked mode:

2.0 Functional Characteristics

53

- The characters ^C, ^Break, ^S, ^P, and ^PrtScrn are handled specially.
- The OS/2 provides buffered input with editing via function keys.
- The data is read until the first ^M or ENTER key is seen. This means that the length of the read data may be less than the requested length. Note that the data is always terminated with the byte sequence 0DH 0AH.
- If ^Z is encountered, no further data is read.
- The data is echoed when read.
- Tabs are expanded into 8-character boundary spaces upon echo, but left as 09H in the buffer.
- * For a write in cooked mode:
 - The ^S is interpreted for flow control.
 - The ^P or ^PrtScrn toggles printer echoing.
 - The ^C or ^Break generates a signal for control-break handling.
 - Tabs are expanded to 8-character boundaries and filled with spaces.
 - Ascii character codes less than 20H are preceded with a caret (^) and 40H is added to the codes.
 - Output is performed up to (but not including) a ^Z or the length of the request. The number actually written may be less than the number requested.

A user process performs I/O to a block device strictly in raw mode. Data is transferred without interpretation or translation.

2.1.4.3 ASCIIZ Strings

Several function calls accept an ASCIIZ string as input. This consists of an ASCII string terminated by a byte of binary zeros.

World Trade Considerations: ASCIIZ strings may be composed of mixed single- and double-byte characters, and may be used in the following cases:

* filename and filename extension

2.0 Functional Characteristics

54

00032

EP 0 415 346 A2

- path name
- directory name

2.1.4.4 Filename Specification

The specification of a filename is dependent on the file system. The OS/2 standard filename consists of drive specifier, pathname, filename and extension. The only required part is the filename. All other parts are optional. The format is as follows - hp2.[D:] [pathname]filename[.ext]

Drive Specifier - hp2.D: This is an optional parameter to specify the logical drive. 'D' can be any drive letter. If this parameter is not specified, the current drive is used. NOTE: This parameter is ignored by the file system if a UNC specifier is also used because UNC names have no associated drive letter.

Pathname Specifier: [prefix][dir1\][dir2\]...[dirn\] This is an optional parameter to specify the directory. prefix: Optional leading '\'. A leading '\'. starts the path from the root directory of the logical drive. If no leading '\'. is specified, the path starts from the current directory of the drive specified (or the current drive if no drive is specified).

A leading '\\'. specifies that this is a UNC name. The entire following name is interpreted by the network software (i.e. \\server\share\dir...\file). UNC paths always start from the root directory of the server share point.

dir\.: Successive directories to be searched. '.' and '..' have special meaning. '.' means the current directory. '..' means the parent directory.

Filename: filename[.ext] This is to specify the filename or directory.

Note: Throughout this text '\'. has been used to denote the path separator character. '\'. and '/' are interchangeable. Some internal and external commands use '/' for specifying options and must have '\'. as the path separator, but the file system does support '/' as a path separator.

2.1.4.5 File Allocation Table (FAT) Type Determination

The FAT file system uses the FAT to map a file's allocation of disk space.

The first two entries in the FAT map a portion of the directory; these two FAT entries contain indicators of the size and format of the disk. The FAT can be in a 12-bit or a 16-bit format. OS/2 (TM) determines whether the disk/diskette has a 12- or a 16-bit FAT format by looking at the total number of allocation units in the disk partition or on the diskette.

OS/2 (TM) determines the type of FAT format (12 or 16-bit entries) by using the following formula:

IF ((TS-RS-(D*BPD/BPS)-(CF*SPF))/SPC)+1 >= 4086
THEN 16-BIT FAT
ELSE 12-bit FAT;

where

- TS = The count of the total sectors on the disk or diskette.
- RS = The number of sectors at the beginning of the disk that are reserved for the boot record. DOS reserves 1 sector.
- D = The number of directory entries in the root directory.
- BPD = The number of bytes per directory entry. BPD is always 32.
- BPS = The number of bytes per logical sector.
- CF = The number of FATs per disk. For most disks CF is 2. For VDISK CF is 1.
- SPF = The number of sectors per FAT.
- SPC = The number of sectors per allocation unit.

The term 'disk' is defined to mean partition for a partitionable device, or that which is accessible through a drive letter.

Fractional cluster numbers that may result from the calculation noted in the above formula are to be rounded downward, i.e., the fractional portion is truncated.

2.1.4.6 Device Names

The operating system has reserved certain names for devices supported by the base device drivers and installable device drivers supplied with the operating system. These device names are listed as follows.

COM1 First serial port.
COM2 Second serial port.
CLOCK\$ Clock.
CON Console keyboard and screen.
SCREEN\$ Screen.
KBD\$ Keyboard.
LPT1 or PRN First parallel printer.
LPT2 Second parallel printer.
LPT3 Third parallel printer.
NUL Nonexistent (dummy) device.
POINTERS\$ Pointer draw device.

These names can be used in the DosOpen system call to OPEN the corresponding devices. Note that these reserved device names take precedence over filenames; the OPEN always checks for a device name BEFORE checking for a filename. This means that a filename which matches a reserved device name can never be OPENed, since the device will be OPENed instead.

Note: There are exceptions to some of these names being reserved. The device driver that supports COM1 and COM2 is installed via CONFIG.SYS. Removing this device driver or replacing it with a device driver that does not support the device names COM1 and COM2, makes COM1 and COM2 no longer reserved in the system. Since the operating system naming convention is for COM1 and COM2 to identify the serial ports, it is not recommended that these names be used for any other purpose.

2.1.4.7 Device I/O Services - General API

2.1.4.7.1 DosBeep - Generate Sound From Speaker:

Purpose Generate sound from speaker.

Format Calling Sequence:

EXTRN DosBeep:FAR

PUSH WORD Frequency ; Hertz
PUSH WORD Duration ; Length of sound
CALL DosBeep

where Frequency is the cycles per second (Hertz) in the range 25H to 7FFFH.

Duration is the length of the sound in milliseconds.

Returns: IF ERROR (AX not = 0)

AX = Error Code

Remarks None.

2.1.4.7.2 DosDevIOctl - I/O Control for Devices:

Purpose Perform control functions on the device specified by the opened device handle.

Format Calling Sequence:

EXTRN DosDevIOctl:FAR

PUSH@ OTHER Data ; Data area
PUSH@ OTHER ParmList ; Command arguments
PUSH WORD Function ; Device function
PUSH WORD Category ; Device category
PUSH WORD DevHandle ; Specifies the device
CALL DosDevIOctl

where Data is a data area.

ParmList is a command-specific argument list.

Function is the device-specific function code.

Category is the device category.

DevHandle is a device handle returned by DosOpen or a standard (open) device handle.

Returns: IF ERROR (AX not = 0)

AX = Error Code

Remarks This function provides a generic, expandable IOCTL facility that replaces and makes obsolete the previous IOCTL system call.

Note: Some IOCTL functions do not require data and/or parameters to be passed when the function is called. For these IOCTLs DosDevIOCTL can be called with a DWORD of zero (null pointer) for either of the ParmList and Data area address fields.

Refer to "-----" on page --- for information regarding specific I/O control commands.

2.1.5 Extended DOS Partition Architecture

The Extended DOS Partition consists of a collection of extended volumes which are linked together by a pointer in the extended volumes' extended boot record. An extended volume consists of an extended boot record and one logical block device. An extended volume created within the extended DOS partition can be any size from one cylinder long up thru the maximum available contiguous space in the extended DOS partition. In OS/2 1.0, an extended volume cannot be larger than 32 MB due to the limitations of the FAT file system. However, in OS/2 1.1 this restriction has been removed from the FAT file system. In OS/2 1.2 partition type 07h has been added for installable file systems. An extended volume can be larger than 32 MB. All extended volumes must start and end on a cylinder boundary. An extended volume will correspond to an image of a physical disk. The extended boot record corresponds to the master boot record at the beginning of an actual physical disk and the logical block device corresponds to the DOS partition that is pointed to by the master boot record.

Therefore the logical block device begins with a normal DOS boot sector if it is a DOS logical block device (syst ind= 1, 4, or 6). IFS logical block

devices (syst ind = 7) need not start with a normal DOS boot sector. This logical block device must start on a track boundary and follow the extended boot record on the physical disk. The logical block device and the extended volume both end on the same cylinder boundary.

Each extended volume will contain an extended boot record, located in the first sector of the disk location assigned to it. This extended boot record will contain the 55AAH signature id byte. This will allow programs that look at the extended (master) boot record to be compatible. This extended boot record will also contain a partition table, which can contain only 2 types of entries. The boot code is not critical, as the devices are not considered bootable. It is suggested that the boot code simply output a message indicating an unbootable partition if it is executed.

The partition table portion of the extended boot records is the same as the partition table structure in the master boot record. This structure has 4 partition entries of 16 bytes each. The system id byte must be filled in for all 4 entries with one of the following values:

01h - DOS partition with SIZE <= 16MB

04h - DOS partition with 16MB < SIZE <= 32MB

05h - maps out area assigned to the next extended volume. Serves as a pointer to the next extended boot record.

06h - DOS partition with SIZE > 32MB

07h - Installable File System

If the system ID byte is 0 then the values in that partition table entry should be zeroed out.

If OS/2 detects any other values other than 01h, 04h, or 06h it should ignore that entry and not attempt to install the logical block device. This will allow future expansion of devices in this area without problems with compatibility with earlier systems.

The partition start and end fields (C,H,S) should be filled in for any of the 4 partition entries in an extended boot record that have one of the above system id bytes. This will allow a program such as FDISK to determine the allocated space in the extended DOS partition, as well as allow the device drivers to determine the physical dasd area that belongs to it. The partition start and end fields (C,H,S) for the partition entry that points to the logical block device (system id 01h, 04h, 06h, or 07h) map out the physical boundaries of the logical block device and are offset relative to the beginning of the extended boot record that the entry resides in. The partition start and end fields (C,H,S) for the partition entry that points to the next extended volume (system id 05h) map out the physical boundaries of the next extended volume and are relative to the beginning of the entire physical disk.

¶ The relative sector and number of sector fields will be set up differently depending on what system id byte is used. If 01h, 04h, 06h or 07h is in the system id field for that extended partition entry (pointer to the logical block device) then the relative sector field should be set up as an offset from (and including) the start of the extended boot record for the associated extended volume. The number of sectors (size) field will be filled in with the size of the created logical block device area, or in other words, the number of sectors mapped out by the start and stop cylinder/track/sector fields. The size of the extended volume can be calculated by adding the relative sector field and the sector size field of the associated extended boot record.

¶ If the system id byte is 05h, then the relative sector field will be the offset (of the NEXT extended volume) in sectors from the start of the entire extended DOS partition. The number of sectors field is not used in this field, and should be filled with 00h's.

¶ This architecture allows only one logical block device to be defined per extended boot record. Therefore, only a maximum of two partition entries at a time will be used in each extended boot record, an entry with system id byte of (01h, 04h, 06h, or 07h) and an entry with id of 05h, which is the pointer to the next extended volume. Although only two entries can be used, a program installing these devices should not assume that the first 2 entries will be the non zero entries.

2.1.5.1 Installing Block Devices in the Extended Partition

¶ To install block devices, the device drivers should first install the primary DOS partitions on all physical drives if any exist. This will insure that an existing drive letter (D:) on the 8th drive will remain the same. After these devices are installed, then on the 8th drive, the drivers should look for the existence of the extended DOS partition. If one exists, then it should look at the first sector of the extended DOS partition for the first extended boot record. If there is a valid system id (01h, 04h, 06h, or 07h) in any of the 4 partition entries, then the device is installed and assigned the next available drive letter. This should occur before any CONFIG.SYS device drivers are loaded to allow FDISK to correctly display the drive letter when space is allocated for the drive.

The first extended boot record (in the extended DOS partition) is a special case in that it is possible that there will not be a device to be installed defined in the partition table. This is because the first device might have been created and then deleted at some time, but the first extended boot record is needed to point to the next one if one exists. Any other extended boot record will always have a device to be installed.

Once a device has been installed (or the special cases above occurs), then the device driver should search the other partition entries for a system id

byte of 05h, indicating that another device (extended volume) exists. If a 05h is not found, then there are no more logical block devices (extended volumes) in the extended DOS partition.

If a 05h system id is found, then the start location in that partition entry should be read in order to find the location of the next extended boot record (extended volume). When it is located, it should be read in and then the process repeated in order to install additional devices.

Once all the valid devices for a physical drive have been installed, then the next physical drive should be examined and the entire process repeated.

A device driver should not assume any order dependency when searching for a particular system id byte in an extended boot record. All four possible entries in a extended boot record partition table should be searched before a driver decides that a particular system id byte does not exist.

¶ The extended DOS partition can only be created if a Primary DOS or IFS partition already exists on a bootable drive. A Primary DOS partition is a partition with a system id byte of 01h, 04h, or 06h. A Primary IFS partition is a partition with a system id byte of 07h. If the drive is not bootable, then an extended DOS partition may be created without having a Primary DOS partition.

The Extended Dos Partition will start and end on a cylinder boundary.

2.1.5.2 Creating Block Devices in the Extended DOS Partition

To create the structure for an extended volume in the extended DOS partition, FDISK should determine if there is available space in the extended DOS partition and if less than 24 total devices are allocated in the system. The maximum number of block devices allowed is 26, and 2 are used by diskettes A: and B:. If so, then the program will create an extended boot record at the space located, with a partition entry filled in with the size and location information for that logical block device. If this is not the first extended boot record, the program should then back up to the last extended boot record in the chain (as linked by the 05h entries), and create an partition entry in that extended boot record that has the size and location data for the newly created one. This action will create the pointer required to locate the boot record just created.

If this is the first extended boot record (in the extended DOS partition), only the size, type and location of the logical block device need to be put into a partition entry. The start of the extended DOS partition in the master boot record will serve as a pointer to this extended volume.

2.1.5.3 Deleting Block Devices in the Extended DOS Partition

To delete a block device, the program should zero out the 16 byte partition entry that contained the system id byte that indicated the device type (01h, 04h, 06h, or 07h). Also, if in the same extended boot record there exists a partition entry with system id of 05h, indicating that another extended volume exists, then this information should be copied to the 05H partition entry of previous extended boot record. There is one exception to this rule. If the logical block device deleted is at the beginning of the extended DOS partition, then only the partition entry indicating the device type would be zeroed out. The 05h pointer information should be left in place.

2.1.5.4 Layout of Block Devices in the Extended DOS Partition

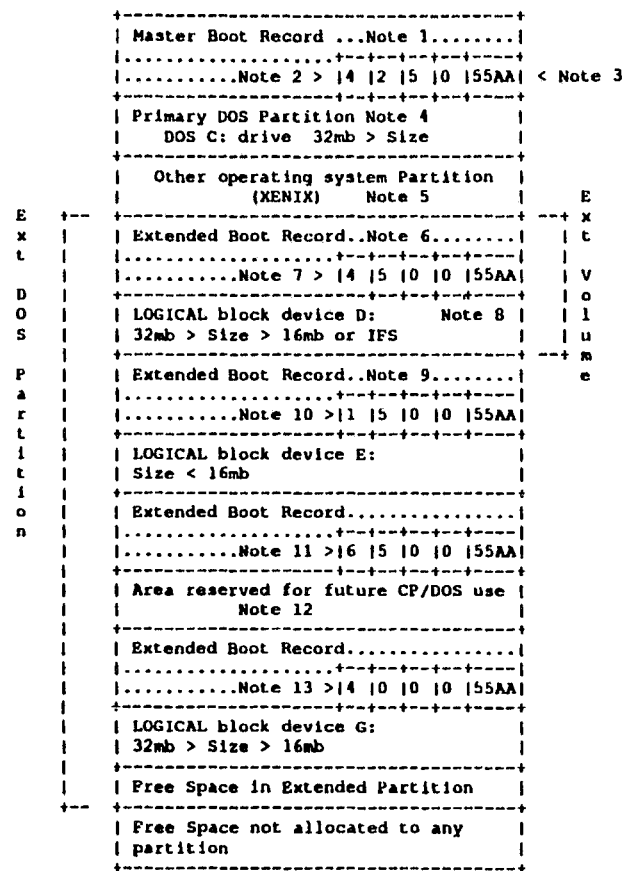


Figure 1. Example of layout of large DASD device with Extended DOS partition.

Note 1 Master boot record code, starting at Trk 000, Hd 00, Sec 01 of disk 80H or 81H.

Microsoft Confidential
OS/2 1.2 IFS Patent Documentation

- Note 2 Partition Table for Master boot record. See IBM PC DOS 3.20 Technical Reference for layout. The 4 is the system id byte in the partition table that indicates a DOS partition where 16MB < SIZE ≤ 32MB, the 2 is a XENIX partition, and the 05h maps the extended DOS partition.
- Note 3 55AAH is the signature to validate the master boot record.
- Note 4 Primary DOS area, must reside entirely in first 32mb of disk. C: is block device 80h, D: is block device 81h if it exists. This partition has a maximum size of 32MB.
- Note 5 Other operating system on disk, XENIX in this example.
- Note 6 Extended boot record for extended volume that corresponds to logical block device D:. (This is assuming only 80h block device exists.) If 81h block device exists, then this would be block device E:.
- Note 7 Logical block device D: partition table entry. This has a max size of 32MB, which is indicated by the system id of 4. This must set the logical DOS block device as starting at the next track boundary. The 05h system id byte in the 2nd partition entry maps out the space allocated to the next extended volume. The starting cyl/sec/head in the partition entry with id of 05h is the location of the next extended boot record of the next extended volume.
- Note 8 Logical block device D:. Logical DOS devices always begin with a DOS boot record as does the primary DOS partition.
- Note 9 Extended boot record for logical block device E:.
- Note 10 Partition table entry for logical block device E:. This logical DOS block device is ≤ 16MB, as indicated by the system ID of 01h. The entry with system id of 05h maps out the space allocated to the next extended volume.
- Note 11 The system id byte of 06h indicates a logical block device > 32MB. This block device is indicated by a block device letter of F. Note also that a pointer exists to the next extended volume.
- Note 12 The >32MB Fat Partition
- Note 13 Partition table entry for final DOS logical block device. Note the absence of 05h id byte means that there are no other extended volumes allocated in the extended DOS partition. This would have a block device letter of G.

Microsoft Confidential
OS/2 1.2 IFS Patent Documentation

2.1.5.5 BPB and Get Device Parameters for Extended Volumes

For purposes of the BPB and Get Device Parameters (Generic IOCTL) an extended volume appears to the system as a virtual physical fixed disk. The extended boot record will correspond to the master boot record of a real fixed disk and the logical block device will correspond to the Primary DOS partition.

This means that the BPB of the logical DOS block device of the extended volume will describe the environment in the extended volume; which consists of the extended boot record and the logical block device. The meaning of the fields will be constant to the meaning of the fields for the Primary DOS partition; as they relate to the entire physical disk, the primary DOS partition, and the master boot record. For example, the number of hidden sectors will be the distance from the beginning of the extended boot record (of the extended volume in question) to the start of the logical DOS block device (the DOS boot record). The number of sectors field will describe only the logical block device just like it normally only describes the primary DOS partition.

2.1.5.6 Category 8 Generic IOCTL Commands

The same philosophy as described above will apply to the disk Generic IOCTL commands. For any logical block device of an associated extended volume; physical Cylinder, Head, Sector I/O will be mapped to within the extended volume. Cylinder 0, Head 0, Sector 1 will be mapped to the extended boot record. An error condition will be generated for any attempt to do C,H,S I/O beyond the size of the extended volume in question.

2.1.5.7 Type 6 Partition

Note the following for a type 6 partition:

- * A 12- or 16-bit type FAT could be used to map it since the type of FAT is strictly based on the number of allocation units (clusters) and is the same algorithm used to define the type of FAT (12 vs 16-bit) in OS/2 1.0.
- * FAT cluster sizes are defined to be based on powers of 2 as for other DOS partition types. Assuming usage of the OS/2 FORMAT utility, the MINIMUM cluster size for a hard file is 2K. Cluster size and the type of FAT (12 vs 16-bit) is determined by the media partition size. The OS/2 FORMAT algorithm is:

```
- If partition size <= 16MB
  then do;
    use 12-bit FAT; /* max 4084 entries */
    max cluster size = 4KB;
  end;
  else do; /* partition size > 16MB */
    use 16-bit FAT; /* max 64K entries */
    min cluster size = 2KB;
  end;
```

Note: however that the actual determination of the partition type is made based on the number of clusters on that partition (OS/2 FORMAT just makes sure that this is true for the <16MB and >16MB partitions).

```
- If number of clusters <= 4084
  use 12-bit FAT; /* max 4084 entries */
  else
    use 16-bit FAT; /* max 64K entries */
```

A partition size of 128MB would require a 2K cluster size based on a maximum 64K allocation units (clusters). A partition size of 129MB-256MB would require a 4K cluster size based on 64K allocation units (clusters). A partition size of 257MB-512MB would require an 8K cluster size based on 64K allocation units (clusters).

Configuration table used by OS/2 FORMAT:

Total # of sectors	Size of partition	Sec/clus	# of root dir entries
32K	16M	8	512
64K	32M	4	512
256K	128M	4	512
512K	256M	8	512
1M	512M	16	512
2M	1G	32	512
4M	2G	64	512
8M	4G	128	512

Please note that for type-6 partitions it is safe to use a non-default configuration, but this may be unsafe for other partition types.

- It can reside any where on the media, as the primary DOS partition

and/or as an extended volume within the extended DOS partition.

- The VPB parameter 'number of sectors per FAT' field width has been extended from a byte to a word in order to define a full 128KB FAT structure. This change affects all DOS partition types.
- In the extended volume context, type 6 > 0 MB.
- In the primary DOS partition context, type 6 > 0 MB. If the partition spans the 32MB boundary or starts at or beyond the 32MB boundary, it can be any size > 0 and STILL BE A TYPE 6 PARTITION.
- On behalf of a DosDevIoctl 'Get Device Parameters' call,
 - If the media is > 32MB, or is defined by a type 6 partition, the recommended BPB must show a 0 in TotalSectors, Ext_TotalSectors must show the number of sectors on the media, and the double word form of 'HiddenSectors' must show the number of sectors after the Master Boot Record to the start of the partition's OS/2 boot record.

2.1.5.8 Layout of Block Devices With A Type 6 Partition

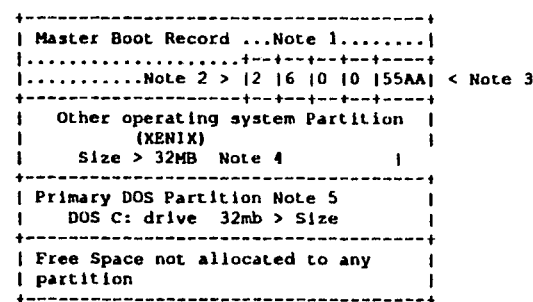


Figure 2. Example of layout of large DASD device with a type 6 partition.

Note 1 Master boot record code, starting at Trk 000, Hd 00, Sec 01 of disk 80H or 81H.

Note 2 Partition Table for Master boot record. See IBM PC DOS 3.20 Technical Reference for layout. The 2 is the system id byte in the partition table that indicates a XENIX partition, and the 06H maps a primary DOS type 6 partition.

Note 3 55AAH is the signature to validate the master boot record.

Note 4 Other operating system (XENIX) on disk.

Note 5 Primary DOS partition. C: is block device 80H, The partition type in this example is a 6 because it ends beyond the first 32MB of the disk. Note that within the scope of this definition that though the size of a primary DOS partition may be less than 32MB, because it ends beyond the first 32MB of the disk it is defined as a type 6.

2.1.5.9 Layout of Block Devices With A Type 6 Partition

```
+-----+
| Master Boot Record ...Note 1.....|
|.....+-----+
|.....Note 2 > |6 |0 |0 |0 |55AA| < Note 3
|.....+-----+
| Primary DOS Partition Note 4      |
|   DOS C: drive  Size > 32mb      |
|.....+-----+
```

Figure 3. Example of layout of large DASD device with type 6 partition.

Note 1 Master boot record code, starting at Trk 000, Hd 00, Sec 01 of disk 80H or 81H.

Note 2 Partition Table for Master boot record. See IBM PC DOS 3.20 Technical Reference for layout. The 6 is the system id byte in the partition table that indicates a DOS partition where 32MB < SIZE.

Note 3 55AAH is the signature to validate the master boot record.

Note 4 Primary DOS area. Owns the entire media and exceeds 32mb in size. C: is block device 80H.

+ 2.1.5.9.1 Type 7 Partition:

+ Note the following for a type 7 partition:

+ * Partition type 7 are used for Installable File Systems only. The internal FAT file system should not use this partition type, since it will mean that older versions of PC-DOS and OS/2 will not be able to access the partition.

2.1.6 Code page support

2.1.6.1 Code Page API

2.1.6.1.1 DosSetCp - Set Code Page:

Purpose DosSetCp allows a process to set its code page and the session's display code page and keyboard code page.

Format Calling Sequence:

```
EXTRN  DosSetCp:FAR  ;
```

```
PUSH  WORD  CodePage ; Code page identifier
PUSH  WORD  Reserved ; Reserved
Call  DosSetCp      ;
```

Where CodePage is a code page identifier word that has one of the following values:

- * 437 IBM PC US 437 code page
- * 850 Multilingual code page
- * 860 Portugese code page
- * 863 Canadian-French code page
- * 865 Nordic code page

Reserved is a reserved word and must be set to zero.

Returns If AX = 0

NO error

ELSE

AX = Error Code:

- * Invalid code page
- * Invalid parameter value

Remarks DosSetCp allows a program to set its code page. See CONFIG.SYS and the CODEPAGE command for preparing code pages for the system. The first code page specified in the CODEPAGE command is the default system code page. The session code page of a new session is set to the default system code page. A session's code page can be changed by the user with the CHCP command at the command

2.0 Functional Characteristics

71

prompt. The process code page of a new program started from a session command prompt is set to that session's code page.

DosSetCp sets the process code page of the calling process. The code page of a process is used in the following ways. First, the printer code page is set to the process code page through the file system and printer spooler (the system spooler must be installed) when the process makes an open printer request. Calling DosSetCp does not affect the code page of a printer opened prior to the call and does not affect the code page of a printer opened by another process. Second, country dependent information will, by default, be retrieved encoded in the code page of the calling process. And third, a newly created process inherits its process code page from its parent process.

DosSetCp also sets, in the session to which the calling process belongs, the code page for the session's default logical keyboard and automatically flushes the keyboard buffer. It also sets the display code page for the session's logical display. This setting of the code page for the session's default logical keyboard and display overrides any previous setting by DosSetCp, KbdSetCp, and VioSetCp by any process in the same session.

The keyboard code page is switched for keyboard handle zero and the keyboard buffer is automatically flushed. The display code page is switched for implicit display handle zero. DosSetCp is for a program to initialize its code page when it begins execution.

DOSSETCP is restricted for use by VIO applications only.

CODE PAGE CONFIGURATION

Code page configuration of the system is necessary to be able to successfully switch between two code pages at run time. The following set of commands must be set up correctly in CONFIG.SYS for this purpose:

Command	Purpose
-----	-----
CODEPAGE	Specify one or two code page identifiers the system is to set up for use.
COUNTRY	Specify the country code and a fully specified file name. The file contains a set of country information in the form of characters encoded according to a code page based on ASCII. The system will default to the

2.0 Functional Characteristics

72

00041

COUNTRY.SYS file in the root directory of the boot drive if no file is specified.

DEVINFO Specify the keyboard layout selection and a fully specified file name. The file contains a keyboard layout table for translating keystrokes into characters encoded according to a code page based on ASCII. The system defaults to the KEYBOARD.SYS file in the root directory of the boot drive if no file is specified.

DEVINFO Specify for the display device a fully specified file name. The file contains a video font table for displaying characters encoded according to a code page based on ASCII. The system does not have a default file name.

DEVINFO Specify for the printer device a fully specified file name. The file contains a printer font table for printing characters encoded according to a code page based on ASCII. The system does not have a default file name.

Incorrect, partial, or mismatched set up of commands for code page selections, country code, keyboard layout, display, and printer may cause ineffective switching between code pages at run time. See the User Reference for the description and syntax of each command and the CHCP change code page command.

Also see DosSetProcCp.

2.1.6.1.2 DosSetProcCp - Set Process Code Page:

Purpose DosSetProcCp allows a process to set its code page.

Format Calling Sequence:

EXTRN DosSetProcCp:FAR ;

PUSH WORD CodePage ; Code page identifier

PUSH WORD Reserved ; Reserved

Call DosSetProcCp ;

Where CodePage is a code page identifier word that has one of the following values:

- * 437 IBM PC US 437 code page
- * 850 Multilingual code page

- * 860 Portugese code page
- * 863 Canadian-French code page
- * 865 Nordic code page

Reserved is a reserved word and must be set to zero.

Returns If AX = 0

NO error

ELSE

AX = Error Code:

- * Invalid code page
- * Invalid parameter value

Remarks DosSetProcCp sets the process code page of the calling process. The code page of a process is used in the following ways. First, the printer code page is set to the process code page through the file system and printer spooler (the system spooler must be installed) when the process makes an open printer request. Calling DosSetProcCp does not affect the code page of a printer opened prior to the call and does not affect the code page of a printer opened by another process. Second, country dependent information will, by default, be retrieved encoded in the code page of the calling process. And third, a newly created process inherits its process code page from its parent process. DosSetProcCp does not affect the display or keyboard code page.

Also see DosSetCp.

2.1.6.1.3 DosGetCp - Get Process Code Page:

Purpose Allows a process to query its current process code page and the prepared system code pages.

Format Calling Sequence:

EXTRN DosGetCp:FAR ;

PUSH WORD Length ; Length of list

PUSH@ OTHER CodePageList ; Address of return data list

PUSH@ WORD Datalength ; Address of return data length

Call DosGetCp ;

Where Length is the byte length of CodePageList.

CodePageList is the return data list where the first word is the current code page identifier of the calling process. If one or two code pages have

Microsoft Confidential
OS/2 1.2 IFS Patent Documentation

been prepared for the system then the second word is the first prepared code page and the third word is the second prepared code page. If the data length is less than the bytes needed to return all the prepared system code pages than the returned list is truncated.

DataLength is the length of the data returned in bytes.

Returns If AX = 0

NO error

ELSE

AX = Error Code:

- * Return data is truncated
- * Address error and data not moved

Remarks

The process code page identifier previously set by DosSetCp or inherited by the process is returned to the caller. An input list size of two bytes returns only the current process code page identifier. If no codepages have been prepared with the CODEPAGE command, a length of two and current codepage identifier value of zero is returned.

The system code page identifiers are returned to the caller in the same order as they appear in the CODEPAGE command. The code page identifiers are returned in the order:

1. The current code page of the process (one of the system code pages).
2. The primary (default) system code page.
3. The secondary system code page, if specified.

2.1.6.1.4 DosGetCtryInfo - Get Country Information:

Purpose Obtains country dependent formatting information that resides in the country file (default name COUNTRY.SYS). The country information returned corresponds to the system country code or selected country code and the process code page or selected code page.

Format Calling Sequence:

EXTRN DosGetCtryInfo:FAR

2.0 Functional Characteristics

75

Microsoft Confidential
OS/2 1.2 IFS Patent Documentation

PUSH WORD Length ;Length of data buffer
PUSH@ OTHER Structure ;Address of data structure
PUSH@ OTHER MemoryBuffer ;Address of return data
PUSH@ WORD DataLength ;Address of return data length
CALL DosGetCtryInfo

Where:

Length is the byte length of the data area (MemoryBuffer) provided by the caller. A length value of 38 bytes is sufficient. In future releases of OS/2 the length required may increase.

Structure is a two word input data structure:

- * Word 0: Country Code
- * Word 1: Code Page ID

Word zero is the binary value of the selected country code where 0 means return the country information for the default system country code. Word one is the binary value of the selected code page identifier where 0 means return the country information for the current process code page of the caller.

MemoryBuffer is the data area where the country dependent information will be placed. This memory area is provided by the caller. The size of the area is provided by the input parameter Length. If it is too small to hold all the available information then as much information as possible is provided in the available space (in the order in which the data would appear). If the amount of data returned is not enough to fill the memory area provided by the caller then the memory that is unaltered by the available data is zeroed out. The format of the information returned in this buffer is:

- 1 Word Country Code.
- 1 Word Reserved (set to zero).
- 1 Word Date format: 0=mm/dd/yy, 1=dd/mm/yy, 2=yy/mm/dd.
- 5 Byte Currency indicator, null terminated.
- 2 Byte Thousands separator, null terminated.
- 2 Byte Decimal separator, null terminated.
- 2 Byte Date separator, null terminated.
- 2 Byte Time separator, null terminated.
- 1 Byte Bit field for currency format:

Bit 0: 1=currency indicator follows money value,

2.0 Functional Characteristics

76

00043

EP 0 415 346 A2

0-currency indicator precedes money value.
Bit 1: Number of spaces (0 or 1) between currency indicator and money value.
Bit 2: When this bit is set, ignore first two bits; Currency indicator replaces decimal indicator.

1 Byte Binary number of decimal places used in currency indication.
1 Byte Time format for file directory presentation:

Bit 0: 1=24 hour
0=12 hour with "a" or "p".

2 Word Reserved (set to zero).
2 Byte Data list separator, null terminated.
5 Word Reserved (set to zero)

DataLength is the length in bytes of the country data returned.

Returns If AX = 0

NO error

ELSE

AX = Error Code:

- * Country information file open failed
- * Country code not found in file
- * Code page identifier not found in file
- * Information type not found in file
- * Buffer too small and information truncated

Remarks None.

2.1.6.1.5 DosCaseMap - Get Case Mapping:

Purpose Performs case mapping on a string of binary values which represent ASCII characters. The case map in the country file (default name COUNTRY.SYS) that corresponds to the system country code or selected country code and the process code page or selected code page is used to perform the case mapping.

Format Calling Sequence:

EXTRN DosCaseMap:FAR

PUSH WORD Length ;Length of String to case map

2.0 Functional Characteristics

77

PUSH@ OTHER Structure ;Address of data structure
PUSH@ OTHER BinaryString ;Address of case map string
CALL DosCaseMap

Where: Length is the byte length of the string of binary values to be case mapped.

Structure is a two word input data structure:

* Word 0: Country Code

* Word 1: Code Page ID

Word zero is the binary value of the selected country code where 0 means use the case map table for the default system country code. Word one is the binary value of the selected code page identifier where 0 means use the case map table for the current process code page of the caller.

BinaryString is the string of binary characters that are to be case mapped. They are case mapped in place and replace the input so the results appear in BinaryString.

Returns: IF AX = 0

NO error

ELSE

AX = Error Code:

- * Country information file open failed
- * Country code not found in file
- * Code page identifier not found in file
- * Information type not found in file

Remarks None.

2.1.6.1.6 DosGetDBCSEv - Get DBCS Environment:

Purpose Obtains a DBCS environmental vector that resides in the country file (default name COUNTRY.SYS). The vector returned corresponds to the system country code or selected country code and the process code page or selected code page.

Format: Calling Sequence:

EXTRN DosGetDBCSEv:FAR

2.0 Functional Characteristics

78

00044

Microsoft Confidential
OS/2 1.2 IFS Patent Documentation

```
PUSH WORD Length      ;length of data buffer
PUSH@ OTHER Structure  ;Address of data structure
PUSH@ OTHER MemoryBuffer ;Address of return data
CALL DosGetDBCSEv
```

Where: Length is the byte length of the data area (MemoryBuffer) provided by the caller. A length value of 10 bytes is sufficient. In future releases of OS/2 the length may increase. The caller can always determine if all the information has been obtained because it terminates with two bytes of zeros. A length of 2 will be sufficient for information returned from non-DBCS related countries.

Structure is a two word input data structure:

- * Word 0: Country Code
- * Word 1: Code Page ID

Word zero is the binary value of the selected country code where 0 means return the DBCS information for the default system country code. Word one is the binary value of the selected code page identifier where 0 means return the DBCS information for the current process code page of the caller.

MemoryBuffer is the data area where the country dependent information for the DBCS environmental vector is returned. This memory area is provided by the caller. The size of the area is provided by the input parameter Length. If it is too small to hold all the available information then as much information as possible is provided in the available space (in the order in which the data would appear). Assuming the data area is large enough, the valid information is terminated by two bytes of 0. The format of the information returned in this buffer is:

2 Byte First range definition for DBCS lead byte values

Byte 1 binary start value (inclusive) for range one
Byte 2 binary stop value (inclusive) for range one

2 Byte Second range definition

Byte 1 binary start value for range two
Byte 2 binary stop value for range two

Microsoft Confidential
OS/2 1.2 IFS Patent Documentation

2 Byte Nth range definition

Byte 1 binary start value for Nth range
Byte 2 binary stop value for Nth range

2 Byte Two bytes of binary zero terminate list

For example: DB 81H,9FH
DB E0H,FCH
DB 0,0

Returns: IF AX = 0

NO error

ELSE

AX = Error Code:

- * Buffer too small, information truncated
- * Country information file open failed
- * Country code not found in file
- * Code page identifier not found in file
- * Information type not found in file

Remarks None.

2.1.6.1.7 DosGetCollate - Get Collating Sequence:

Purpose Obtains a collating sequence table (for characters 00H through FFH) that resides in the country file (default name COUNTRY.SYS). It is used by the SORT utility to sort text according to the collating sequence. The collating table returned corresponds to the system country code or selected country code and the process code page or selected code page.

Format: Calling Sequence:

EXTRN DosGetCollate:FAR

```
PUSH WORD Length      ;length of data buffer
PUSH@ OTHER Structure  ;Address of data structure
PUSH@ OTHER MemoryBuffer ;Address of return data
PUSH@ WORD DataLength  ;Address of return data length
CALL DosGetCollate
```

Where Length is the byte length of the data area (MemoryBuffer) provided by the caller. A length value of 256 bytes is sufficient. In future releases of OS/2 the length may increase.

Structure is a two word input data structure:

- * Word 0: Country Code
- * Word 1: Code Page ID

Word zero is the binary value of the selected country code where 0 means return the collate table for the default system country code. Word one is the binary value of the selected code page identifier where 0 means return the collate table for the current process code page of the caller.

MemoryBuffer is the data area where the collating sequence table is returned. This memory area is provided by the caller. The size of the area is provided by the input parameter length. If it is too small to hold all the available information then as much information as possible is provided in the available space (in the order in which the data would appear). If the amount of data returned is not enough to fill the memory area provided by the caller then the memory that is unaltered by the available data is zeroed out. The format of the information returned in this buffer is:

1 Byte Sort weight of ASCII (0)
1 Byte Sort weight of ASCII (1)
... (additional values in collating order)
1 Byte Sort weight of ASCII (255)

DataLength is the length in bytes of the collate table returned.

Returns: IF AX = 0

NO error

ELSE

AX = Error Code:

- * Country information file open failed
- * Country code not found in file
- * Code page identifier not found in file
- * Information type not found in file
- * Buffer too small and information truncated

Remarks None.

2.1.7 System Initialization and Configuration

- * The purpose of system initialization is to establish the environment in which OS/2 (TM) executes. However, options exist in the system environment that the user may choose to configure.

2.1.7.1 System Initialization

System initialization is done either as a result of a power on or by a system reset. System initialization is accomplished by the following:

1. The Boot Sector
2. OS/2 (TM) Device Interface Module initialization routine
3. OS/2 (TM) Kernel Module initialization routine
4. OS/2 (TM) System Initialization Process

2.1.7.1.1 The Boot Sector:

For diskettes, the boot sector begins on track 0, sector 1, side 0. For fixed disks, the boot sector begins on the first sector of the OS/2 (TM) partition.

At a power on or system reset, ROM BIOS is invoked and performs hardware checks and initialization. Then ROM BIOS examines drive A for the boot sector. If the boot sector is not found, ROM BIOS then looks in the active partition of the fixed disk. If no boot sector is found, then ROM BIOS invokes ROM BASIC. However, if a boot sector is located, ROM BIOS reads it into low memory and gives it control.

When the Boot code gains control, the mode of operation is Real Mode.

The Boot code loads the OS/2 (TM) Device Interface Module into low memory and gives it control.

2.1.7.1.2 OS/2 (TM) Device Interface Module:

The OS/2 (TM) Device Interface Module invokes its initialization routine. This initialization routine performs an equipment check and loads the OS/2 (TM) Kernel Module. It then invokes the OS/2 (TM) Kernel Module.

2.1.7.1.3 OS/2 (TM) Kernel Module:

The OS/2 (TM) Kernel Module invokes its initialization code. This initialization code relocates the code for the System Initialization Process into high conventional memory (640 Kb or less). It also relocates kernel code and data segments in the appropriate places in memory. It initializes kernel components as well as the default device drivers. The scheduler is initialized with the System IDLE Process and the System Initialization Process. The mode of operation is set for protect mode and the System Initialization Process is invoked.

2.1.7.1.4 OS/2 (TM) System Initialization Process:

The System Initialization Process handles the configuration commands in the CONFIG.SYS file, establishing the final operating environment.

2.1.7.2 System Configuration (CONFIG.SYS)

CONFIG.SYS is the file that contains commands used to configure the system. Only a single CONFIG.SYS file is needed to configure the system for both real mode and protect mode operations.

Configuration commands and parameters will conform to the guidelines for country dependent information and messages as discussed in "-----" on page ---.

During system initialization, OS/2 (TM) opens and reads the CONFIG.SYS file in the root directory of the drive from which it was started and interprets the commands within the file.

If a keyword specification is invalid it is ignored and the System Initialization default value is used. Reference each config.sys command for the default values. If a keyword is specified multiple times, the last valid specification is used except for the following keywords which may be specified multiple times:

* DEVICE

NOTE: The System Install component places several files in 1 or more directories generating the appropriate config.sys commands, thus making the CONFIG.SYS file a required file to successfully execute OS/2 (TM). Reference the System Install component for the list of commands required for the CONFIG.SYS file.

The following list summarizes the configuration commands for OS/2 (TM).

AUTOFAIL Disable/Enable system wide hard error and exception popup.

2.0 Functional Characteristics

83

BUFFERS Determine the number of buffers to allocate for disk I/O.
COUNTRY Select the format for country-dependent information.
DEVICE Specify path and filename of a device driver to be installed.

2.1.7.3 OS/2 (TM) Configuration Command Descriptions

2.1.7.3.1 AUTOFAIL:

Purpose Disable/Enable system wide hard error and exception popup.

Format AUTOFAIL = yes | no

where yes means that hard error and exception popups will not occur. Whenever such condition is encountered, an appropriate error code will be return instead.

no means that hard error and exception conditions will cause a popup to occur.

Remarks The default is NO so the system will display the popup menu when a hard error or exception condition occurs.

2.1.7.3.2 BUFFERS:

Purpose Determines the number of buffers used for caching disk I/O that OS/2 (TM) will allocate.

Format BUFFERS = x

where x is a number between 1 and 100.

Remarks The default value is 3.

The disk buffer is a block of memory that a file system may use to hold data being read from or written to a disk. Before the file system reads or writes a record, it checks to see if that record is contained in a sector already in a buffer. If so, then the file system does not need to access the disk.

The resident size of OS/2 (TM) increases by 512 bytes for each additional buffer specified.

2.0 Functional Characteristics

84

00047

2.1.7.3.3 COUNTRY:

Purpose Selects the format for the country-dependent information.

- * Date and Time format
- * Currency symbol
- * Decimal separator

Format COUNTRY = nnn[, [d:] [path] filename.ext]

where nnn is the 3-digit international country code for the telephone system. Refer to "-----" on page --- for details and the list of supported countries.

Remarks The default value is the U.S. country code of 001.

If filename is not specified, the country information is supplied by the default COUNTRY.SYS file in the root directory on the system reset volume.

If a filename is specified, the default drive/path is the system root directory if the drive/path is not specified.

2.1.7.3.4 DEVICE:

Purpose Specifies the path and filename of a device driver in order that it may be installed.

Format DEVICE = [d:] [path] filename [.ext] [arguments]

where d: is the drive and is optional
path is optional
filename.ext is required
arguments are dependent on the device driver

Remarks The standard default device drivers loaded by OS/2 (TM) support the console, printer, diskette, fixed disk, and clock devices. These device drivers do not require any DEVICE= entries in CONFIG.SYS.

To install other device drivers, the DEVICE= entry is required.

Both old and new device drivers may be loaded with the DEVICE= command. New OS/2 (TM) device drivers are initialized in protect mode; old device drivers are initialized in real mode.

DEVICE= commands are processed in the order in which they appear in CONFIG.SYS.

No DEVICE= commands (for model group dependent device drivers) will be allowed in CONFIG.SYS to support basic OS/2 operation. If a default CONFIG.SYS is shipped with OS/2 then it will contain no DEVICE= commands for device drivers that are model group dependent. Any device drivers required for OS/2 basic operation will be loaded in automatically from hidden and reserved file names. The filenames used will depend on the model group that the IPL is occurring on.

The correct DEVICE= parameters are determined by INSTALL or the user as discussed above.

2.1.8 Installable File System

2.1.8.1 Requirements on The IFS Mechanism

The Installable File System (IFS) Mechanism supports the following:

- * Coexistence of active file systems in a single PC,
- * Multiple logical volumes (partitions),
- * Multiple and different storage devices,
- * Redirection or connection to remote file systems,
- * File system flexibility in managing its data and I/O for optimal performance,
- * Transparency at both the user and application level.
- * Standard set of File I/O API,
- * Existing logical file and directory structure,
- * Existing naming conventions,

- File system doing its own buffer management,
- File system doing file I/O without intermediate buffering,
- Extensions to the Standard File I/O API (File System CTL),
- Extensions to the existing naming conventions,
- IOCTL type of communication between a file system and a device driver.

2.1.8.2 Description

2.1.8.2.1 System Relationships:

The IFS Mechanism defines the relationships among the operating system, the file systems, and the device drivers. The basic model of the system is represented in Figure 4 on page 88.

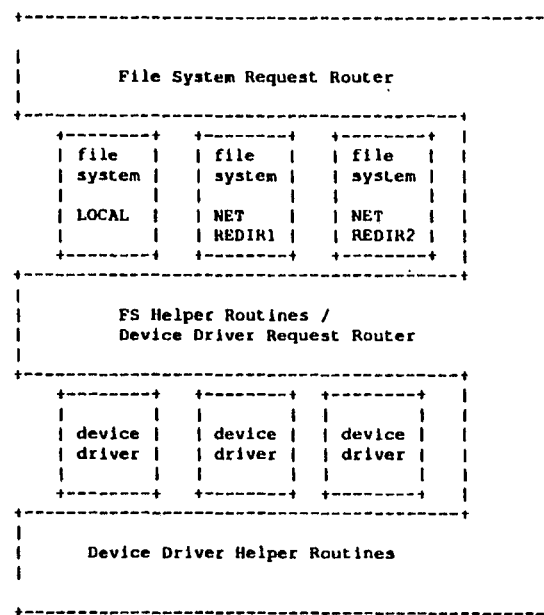


Figure 4. System Relationships for Installable File Systems

The Request Router directs File I/O system calls to the appropriate file system for processing.

The file systems manage file I/O and control the format of information on the storage media. An installable file system will be referred to as a file system driver or FSD.

The FS Helper Routines provide a variety of services to the file systems.

The device drivers manage physical I/O with devices. Device drivers do not understand the format of information on the media.

2.1.8.2.2 Standard File I/O:

Standard file I/O is performed through the Standard File I/O API. The user makes a system call, and the Request Router passes the request to the correct file system for processing.

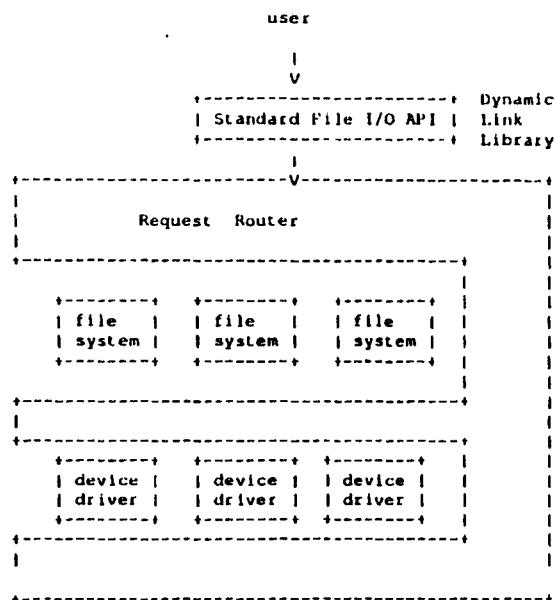


Figure 5. Standard File I/O

2.1.8.2.3 Extended File I/O:

New API may be provided by a file system to implement functions specific to the file system or not supplied through the standard file I/O interface. Any new API would be provided as a dynamic link package that would use the DOSFSCTL to communicate with the specific file system.

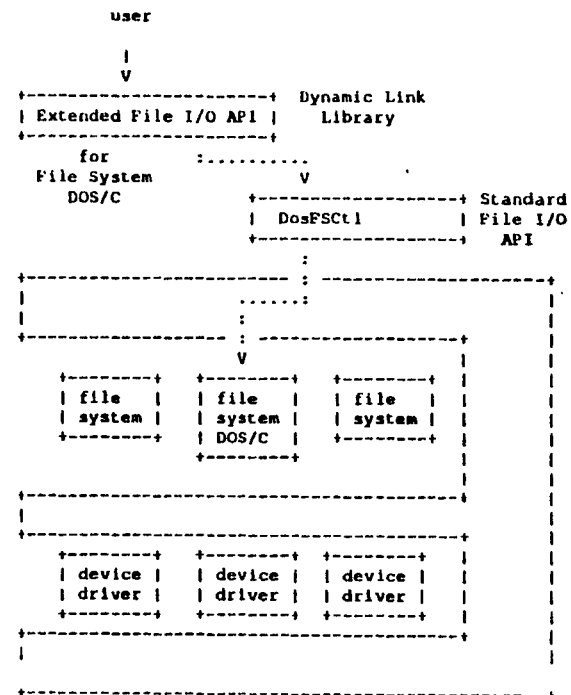


Figure 6. Extended File I/O

As can be seen by the examples, the file system does not have special knowledge of the nature of the task's original request. One reason for

The IFS mechanism allows an FSD to do its own buffer management.

The FSD moves all data, requiring partial sector I/O, between the application's buffers and its cache buffers. The FS helper routines initiate the I/O for local file systems.

An FSD may use the OS/2-managed buffer cache using FS helpers.

An FSD may do all buffering entirely on its own.

- * Volume management, i.e., detecting when the wrong volume is mounted and notifying the operator to take corrective action, is handled directly through OS/2 and the device driver. Each FSD is responsible for generating a volume label and 32-bit volume serial number. It is strongly suggested that these be stored in a reserved location in logical sector zero at format time.

- * It is not required that an FSD use a particular format to store this
- * information. OS/2 will call the FSD to perform operations that might
- * involve it. The FSD is required to update the VPB whenever the volume label
- * or serial number is changed.

When the FSD passes an I/O request to an FS helper routine the driver passes the 32-bit volume serial number and the user's volume label (via the VPB). When the I/O is performed, OS/2 compares the requested volume serial number with the current volume serial number it maintains for the device. This is an in-storage test (no I/O required) performed by checking the Drive Parameter Block's (DPB) VPB of volume mounted in the drive. If unequal, OS/2 signals the critical error handler to prompt the user to insert the volume having the serial number and label specified.

- ```
* When OS/2 detects a media change in a drive, or the first time a drive is
* accessed on behalf of an API function call, it will determine the FSD (file
* system driver) that will be responsible for managing I/O to that volume.
* OS/2 will allocate a VPB (volume parameter block) and poll the installed
* FSDs (by calling the FS_MOUNT entry point) until an FSD indicates that it
* does recognize the media.
```

- \* The FAT FSD will be the last in the list of FSDs and, by recognizing all media, will act as the default FSD when no other FSD recognition takes place.

#### 2.1.8.2.7 Connectivity:

There are two classes of file system drivers:

- \* an FSD which uses a block device driver to do I/O to a local or remote (virtual disk) device. This is called a local file system.
- \* an FSD which accesses a remote system without a block device driver. This is called a remote file system.

The connection between a drive letter and a remote file system is achieved through a command interface.

The connection between a pseudo-character device and a local or remote file system is achieved through a command interface.

- \* When a local volume is first referenced OS/2 sequentially asks each FSD in the FSD chain to accept the media, via call to each FSD's FS\_MOUNT entry point. If no FSD accepts the media then it is assigned to the default FAT file system. Any further attempt made to access an unrecognized media other than by FORMAT, will result in an 'Invalid media format' message.

- \* Once a volume has been recognized, the relationship between drive, FSD, volume serial number, and volume label is remembered. The volume serial number and label are stored in the Volume Parameter Block, (VPB). The VPB is maintained by OS/2 for open files (file-handle based I/O), searches, and buffer references. The VPB represents the media.

- \* Subsequent requests for a removed volume require polling the installed FSDs for volume recognition by calling FS\_MOUNT. The volume serial number and volume label of the VPB returned by the recognizing FSD and the existing VPB are compared. If the test succeeds, the FSD is given access to the volume. If the test fails, OS/2 signals the critical error handler to prompt the user for the correct volume.

The connection between media and VPB is remembered until all open files on the volume are closed, search references and cache buffer references are removed. Only volume changes cause a re-determination of the media at the time of next access.

#### 2.1.8.2.8 IPL Mechanism:

- \* A primary DOS disk partition (type 1, 4, or 6) may be used to boot the system. The code for FSDs may reside in any partition readable by a previously installed FSD. An IFS partition must be a type 6.

The OS/2 boot partition includes the following: Boot record, basic file system structure, BIOS and DOS files, OS/2 initialization files, Advanced BIOS patch files, CONFIG.SYS, and base device drivers. Boot partitions always contain FAT file systems.

#### 2.0 Functional Characteristics

95

If the media isn't bootable, there need not be a FAT partition present.

Device drivers and FSDs are loaded in the order they appear, and are considered elements of the same ordered set. Thus both device drivers and FSDs may be loaded from installed file systems, so long as they are started in the proper order. For example:

```
DEVICE = c:\devices\diskdrv.sys
REM Block device D: is now defined. (diskdrv.sys controls this.)
IFS = c:\fsd\newfs1.fsd
REM If we assume that D: contains a fixed newfs1 type partition,
REM then we're now ready to use D: to load the device driver and
REM FSD for E:
DEVICE = d:\root\dev\special.dev
REM Block device E: is now defined.
IFS = d:\root\fsd\special.fsd
REM E: can now be read.
DEVICE = e:\music
```

#### 2.1.8.2.9 OS/2 Partition Access:

Access to the OS/2 partition on a bootable, logically partitioned media is through the full OS/2 function set. A detailed description of the disk partitioning design is available in the OS/2 1.1 FPFS.

#### 2.1.8.2.10 Permissions:

An architecture for secure file systems has been considered but not formalized. There are no secure file system clients identified for the first release of OS/2 incorporating the IFS architecture.

#### 2.1.8.2.11 File Naming Conventions:

- \* OS/2 1.2 will view 'path names' as ASCII2 strings and not restrict FSDs to the DOS file name format: xxxxxxx[.yyy] (8.3 format). This allows an FSD to use whatever naming convention is appropriate for it today and to extend that naming convention in future releases as necessary. DOSQSYSINFO should be called at initialization time to determine maximum path length.

- + Applications that can support non-8.3 format filenames must set the LONGNAMES bit in their executable header in order to be able to manipulate such names.

After the OS/2 name processing has completed, the path is passed to the FSD using the FS\_PROCESSNAME interface for enforcement of FSD-specific naming conventions.

'\' and '/' are not valid file name characters. Names in paths are

#### 2.0 Functional Characteristics

96

00053

delimited by '\' or '/' and have no drive letters. This supports other Operating System name formats (e.g., VM, UNIX, RPS, EDX).

The file names '.' and '..' are reserved for use by hierarchical directory structures for the current directory and the parent of the current directory respectively. The names '.' and '..' receive special processing during canonicalization, and will never be seen by any FSD.

The '.' and '..' notation must be used by file systems using hierarchical directory structures, with the semantics described above. In addition, '.' and '..' must be physically present in all directories.

File system drivers which support hierarchical directory structures must use '\' and '/' as path name component separators. '\' and '/' are identical in meaning. No other character may be accepted as a path separator. File system drivers which do not support hierarchical directory structures must reject as illegal any use of '\' or '/' in path names.

- \* If an FSD uses a component separator within a filename, it must be '.'.
- \* There are no restrictions on the number of components which may be allowed within a file name.

- \* FSD filenames are restricted to the OS/2 character set.

All filenames are case insensitive (that is, all lower case alphabetic characters are automatically converted to the equivalent upper case characters). This applies to all system canonicalized file names. The filenames passed to an FSD will be converted to uppercase but the filenames/data that may be passed in data buffers to an FSD will not be processed this way.

The invalid characters for filenames are the range 0-1Fh and the characters '\', '/', '\'', '[', ']', ':', '<', '>', '+', '=', ';', and ','. This list applies to labels and EA names also.

For compatibility reasons, the OS/2 FAT file system will continue to only support the old DOS file name format: xxxxxxxx[.yyy] (8.3 format). It is required that OS/2 and PC/DOS media be compatible within the FAT File System context.

Compatibility with existing DOS 3 applications requires all FSDs to support a superset of the FAT file system's 8.3 format.

IFS requests issued from the 3x box will have the 8.3 truncation rules applied before the FSD receives the request. FSDs do not need to know if a particular request is issued by the 3x box.

Applications are not expected to understand FSD-specific naming conventions. Parsing of names is to only take into account '\', '/', '.', and '..'.

- \* For compatibility reasons, trailing dots on component names are discarded.
- \* For example, "\foo.bar...bletch...\a..b...\c." becomes

+ "\foo.bar...bletch\a..b\c". This processing includes semaphore, queue, pipe, module, shared memory names, and device names.

#### \* 2.1.8.2.12 Meta Character Processing:

Meta characters will work in the following way for all programs regardless of level number. Meta characters are illegal in all but the last component of a path.

\* Meta characters have two sets of semantics. The first is as search characters, where they are used to select which filenames are returned to the user. The second is as edit characters, used to construct a new name given a source name and a target name specification. The second use occurs in places like "copy \*.txt \*.old", and is not actually supported anywhere in the kernel except in FCB\_rename, and some special FCB editing calls.

\* Thus, both "\*" and "?" have two rules, one for searching, one for copy-editing used in "ren <name-with-metas> <name-with-metas>", and such.

\* "." has no special meaning itself in searching, but ? gives it one.

\* "." has a special meaning for editing.

#### \* Searching

- \* \* matches 0 or more characters, any character, including blank. It will not cross NUL or \. (Which means it only matches a filename, not an entire path.)
- | ? matches 1 character, unless what it would match is a '.' or the terminating NULL, in which case it matches 0 characters. (It also doesn't cross \.)
- \* Any character other than \* and ? matches itself, including .

#### \* Editing

- \* metas in the source simply match files, and behave just like any other search meta. (see above)
- \* metas in the target are copy-edit commands, and work like this:
  - \* ? copies one character, unless what it would copy is a ".", in which case it copies 0. It also copies 0 characters if we're at the end of the source string.
  - | \* copies characters from the source to the target until it finds a source character that matches the character following it in the target.
  - \* . in the target "syncs" pointers. It causes the source pointer to match the corresponding "." in the target. They count from the left.
- \* The DOSEDTNAME API performs the described operation.

+ For compatibility reasons, any file name that does not have a dot in it gets an implicit one automatically appended to the end during searching operations. This means that searching for "foo." would return "foo".

#### 2.1.8.2.13 Family API issues:

Since the IFS mechanism is neither present in previous releases of OS/2 nor present in real-mode DOS versions 2.0 through 3.3, FAPI will not be extended to support the new interfaces.

#### | 2.1.8.2.14 FS Utility Support:

| Each FSD is required to provide a single .DLL executable module in order to support the OS/2 FORMAT, CHKDSK, SYS, and RECOVER utilities. The FS-provided executable will be invoked by these utilities when performing a FORMAT, CHKDSK, SYS, or RECOVER function for that file system. The command line arguments and environment pointer that was passed to the utility will be passed unchanged to the FS-specific executable.

| The procedures will reside in the file called U<fsdname>.DLL, where <fsdname> is the name returned by DosQFsAttach. If the file system wishes to be able to have its utility support .DLL file on a FAT partition, then it should choose <fsdname> to be up to 7 bytes long.

| The FSD utility procedures are expected to follow the following rules:

- | \* No preparation will be done by the base utilities before they invoke the FSD utility procedure. Thus, the base utilities will not lock drives, parse names, open drives, etc. This allows maximum flexibility for the FSD author.
- | \* The FSD utility procedures are expected to follow the standard conventions for the operations that they are performing, e.g. /F for CHKDSK implies "fix".
- | \* The FSD procedures may use stdin, stdout and stderr if they wish, but should be aware that they may have been redirected to a file or device.
- | \* It is the responsibility of the FSD procedures to worry about volumes being changed while the operation is in progress. The normal action would be to stop the operation when such a situation is detected.
- | \* When the FSD procedures are called, they will be passed argc, argv and envp that they can use to determine the operations.
- | \* The FSD procedures are responsible for putting out the relevant prompts and messages.
- | \* The FSD utility procedures are expected to follow the standard convention of entering the target drive as specified for each utility.

| Interfaces for the FSD procedures

| All the FSD utility procedures are called with the same arguments as follows:

- | \* int far pascal Ufsdname.CHKDSK(int argc, char far \* far \*argv, char far \* far \*envp)
- | \* int far pascal Ufsdname.FORMAT(int argc, char far \* far \*argv, char far

```
| * far *envp)
|
| * int far pascal Ufsdname.RECOVER(int argc, char far * far *argv, char far
| * far *envp)
|
| * int far pascal Ufsdname.SYS(int argc, char far * far *argv, char far *
| far *envp)
```

| Where argc, argv and envp have the same semantics as the corresponding  
| variables in C.

#### 2.1.8.2.15 FSD Pseudo-Character Device Support:

A pseudo-character device (single file device) may be redirected to an FSD. The behavior of this file is very similar to the behavior of a normal DOS character device. It may be read from (DosRead) and written to (DosWrite). The difference is that the DosChgFilePtr and DosFileLocks functions can also be applied to the file. The user would perceive this file as a device name for a non-existing device. This file is seen as a character device because the current drive and directory have no effect on the name. This is what happens in DOS today for character devices. For example:

##### FILESYS HOST BCRVMPC1 USERID PASSWORD

would define a file (or device) through which host communication can be established. A TYPE HOST command would display the latest host created data and a COPY CON HOST command could be used to send commands to the host. This example assumes the FSD BCRVMPC1 can perform the necessary host communication and translation.

The format of an OS/2 pseudo-character device name (i.e., single file device) is that of an ASCII string in the format of a OS/2 filename in a subdirectory called \DEV\ . The pseudo device name HOST is accessible at the API level (DosQFSAttach) through the path name: '\DEV\HOST'.

#### 2.1.8.3 Extended attributes

Extended attributes (EAs) are a mechanism whereby an application can attach information to a file system object (directories or files) describing the object to another application, to the operating system, or to the FSD managing that object.

This data is not kept as part of the object's data itself; to do this would require all applications to understand the object's format and would give the applications direct access to all other EAs. Normally, applications

will deal with a subset of the EAs that may exist on an object.

Each EA has two parts: a name and a value. The name is ASCII text that is used to identify a particular EA. There is no other mechanism to identify a particular EA. The name portion of EAs is chosen by the application programmer. This poses the problem of name collision. To address this, a naming convention is adopted. This convention is quite simple; the name is prefixed with the name of the company (or suitable abbreviation) and the name of the product (or suitable abbreviation) that defines the attribute.

All kernel-defined EAs will be prefixed with "DOS.".

\* EA names are restricted to the same character set as filenames.

The value portion of an EA may be arbitrary data or it may be ASCII data. There are many different representations for the value portion of EAs. Rather than mandate a specific format, the following guidelines should be used:

If a simplification/optimization can be achieved in the application by representing the value as binary data (such as icons or timestamps) then the representation should be binary.

Otherwise, the representation should be displayable in all character sets (i.e. restricted to characters 0x20 through 0x7E).

EAs may be viewed as a property list attached to objects. The services for manipulating EAs are: add/replace a series of name/value pairs, return name/value pairs given a list of names, and return the total set of EAs.

There are two formats for EAs as passed to OS/2 v1.2 API: Full EAs (FEA) or Get EAs (GEA).

FEAs are complete name/value pairs. In order to simplify and speed up scanning and processing of these names, they are represented as length-preceded data. FEAs are defined as follows:

```
struct FEA {
 unsigned char reserved; /* must be 0 */
 unsigned char cbName; /* length of name */
 unsigned short cbValue; /* length of value */
 unsigned char szName[]; /* ASCII name */
 unsigned char aValue[]; /* free-format value */
};
```

\* The name length does not include the trailing NUL. The maximum EA name length is 255 bytes. The minimum EA name length is 1 byte. The characters that form the name are legal filename characters. Wildcard characters are not allowed. EA names are case-insensitive and should be uppercased. The

\* FSD should call FSH-CHECKEASNAME and FSH\_UPPERCASE for each EA name it receives to check for invalid characters and correct length, and to uppercase it.

A list of FEAs is a packed set of FEA structures preceded by a length of the list (including the length itself) as indicated in the following structure:

```
struct FEAList {
 unsigned short cbList; /* length of list */
 struct FEA list[]; /* packed set of FEA */
};
```

FEA lists are used for adding, deleting, or changing EAs. A particular FSD may store the EAs in whatever format it desires. Certain EAs may be stored to optimize retrieval.

A GEA is an attribute name. Its format is:

```
struct GEA {
 unsigned char cbName; /* length of name */
 unsigned char szName[]; /* ascliz name */
};
```

The name length does not include the trailing NUL.

A list of GEAs is a packed set of GEA structures preceded by a length of the list (including the length itself) as indicated in the following structure:

```
struct GEAList {
 unsigned short cbList; /* length of list */
 struct GEA list[]; /* packed set of GEA */
};
```

GEA lists are used for retrieving the values for a particular set of attributes. It is used as input only.

Name lengths of 0 are illegal and are considered in error. A value length of 0 has special meaning. Setting an EA with value length of 0 will cause that attribute to be deleted (if possible). Upon retrieval, a value length of 0 indicates that the attribute is not present.

Setting attributes contained in an FEA list treats the entire FEA list as atomic; all of the changes specified in the list are applied or none will. This is required since the attributes together may specify some behaviour for the file system and may not make sense being set independently.

The routing of EA requests is accomplished via the IFS routing mechanism. EA requests that apply to names are routed to the remote FSD attached to the specified drive or to the local FSD managing the volume in the named drive. Those requests that apply to a handle (file or directory) are routed to the FSD attached to the handle. No interpretation of either FEA lists nor GEA lists is performed by the IFS router. It is the responsibility of each FSD to provide support for EAs. It is expected that some FSDs will be unable to store EAs (e.g. UNIX- and MVS-compatible file systems).

The FAT FSD implementation will provide for the complete implementation of EAs. There will be no special EAs for the FAT FSD.

All EA manipulation is performed using the following structure; the relevance of each field is described within each API:

```
struct EAOP {
 struct GEAList far * fpGEAList; /* GEA set */
 struct FEAList far * fpFEAList; /* FEA set */
 unsigned short offError; /* offset of FEA err */
};
```

\* In the OS2 V1.2 release, values of cbList greater than (64K-1) will not be allowed. This is an implementation defined limitation which may be raised in the future. Because this limit may change, programs should avoid enumerating the list of all EAs, but instead manipulate only EAs that they now about. For operations such as copying, the DosCopy api should be used. If enumeration is necessary, the DosEnumAttribute API should be used.

\* A special category of attributes, called create-only attributes, is defined as the set of extended attributes that a file system may only allow to be set at creation time. (Such attributes may be used to control file allocation and structure configuration.) File systems are expected to allow create-only attributes to be set at any time >from when the object is created to when it is first modified, i.e. data written into a file or an entry added to a directory. Programs that copy objects should copy all of the EAs for an object before otherwise modifying it in order to assure that any create-only attributes from the source are properly applied to the target. The DosCopy() api is the preferred way for copying files or directories.

#### 2.1.8.3.1 FSD File Image:

An FSD loads from a file which is in the format of a standard OS/2 dynamic link library file. Exactly one FSD resides in each file. The FSD exports information to OS/2 using a set of predefined public names.

The FSD is initialized by a call to the exported entry point FS\_INIT.

FS entry points for Mount, Read, Write, etc. are exported with known names

In addition to various entry points, the FSD must export a dword bit vector of attributes. Attributes are exported under the name 'FS\_ATTRIBUTE'. FS\_ATTRIBUTE specifies special properties of the FSD and is described in the next section.

▪ The format of the OS/2 FS ATTRIBUTE field is defined below:

| BITS                          | DESCRIPTION                                                        |
|-------------------------------|--------------------------------------------------------------------|
| * 31                          | FSD_Extended_Attributes. If 1, FSD has extended attributes. If     |
| * 0,                          | FS_ATTRIBUTE is only FSD attribute information.                    |
| * 30-28                       | VERSION_NUMBER - FSD version number.                               |
| 27-3                          | RESERVED.                                                          |
| + 2                           | FIO - File I/O bit. Set if FSD wants to see file locking/unlocking |
| + operations                  | and compacted file I/O operations. If not set, the                 |
| + fileio                      | calls will be broken up into individual                            |
| + lock/unlock/read/write/seek | calls and the FSD will not see the                                 |

REM - remote file system. This bit tells the system whether the FSD uses static or dynamic media attachment. Local FSDs always use dynamic media attachment. Remote FSD's always use static media attachment. This bit is Clear if it is a dynamic media attachment and Set if static attachment. No FSD supports both static and dynamic media attachment. To support proper file locking, a remote FSD should also set the FIO bit.

FSD initialization occurs at system initialization time. FSDs are loaded via the IFS = configuration command in CONFIG.SYS. Once the FSD has been loaded, the FSD's initialization entry point is called to initialize it.

OS/2 FSDs initialize in protect mode. Because of the special state of the system, an FSD may make dynamic link system calls at init-time. The list of system calls that an FSD may make are as follows:

DosBeep  
DosChgFilePtr  
DosClose  
DosDelete  
DosDevConfig  
DosDevIoCtl  
DosFindClose  
DosFindFirst  
DosFindNext  
DosGetEnv  
DosGetInfoSeg  
DosGetMessage  
DosOpen  
DosPutMessage  
DosQCurDir

Microsoft Confidential  
OS/2 1.2 IFS Patent Documentation

```

* DosQCurDisk
* DosQFileInfo
* DosQFileMode
* DosQSysInfo
* DosRead
* DosWrite

```

The FSD may not call FS helpers at initialization time.

It should be noted that multiple code and data segments are not discarded by the loader as in the case of device drivers.

- \* The FSD may call DosGetInfoSeg to obtain access to the global and process local information segments. The local segment may be used in the context of all processes without further effort to make it accessible and has the same selector. The local infoseg is not valid in real mode or at interrupt time.

#### 2.1.8.4.1 OS/2 and PC/DOS 3.4 Boot Record and BIOS Parameter Block:

The extended Boot structure:

```

Extended_Boot struct
Boot_jmp db 3 dup (?)
Boot_OEM db 8 dup (?)
Boot_BPB db (size Extended_BPB) dup (?)
Boot_DriveNumber db ?
Boot_CurrentHead db ?
Boot_Sig db 41 ;Indicate Extended Boot structure.
Boot_Serial dd ?
Boot_Vol_Label db 11 dup (?)
Boot_System_ID db 8 dup (?); "FAT", "OTHER_FS"
Extended_Boot ends
Area for code and messages
Boot_Signature db 55h, 0AAh

```

Where Serial is the 32-bit binary volume serial number for the media.

System\_ID is an 8-byte name written when the media is formatted. It is used by FSDs to identify their media, but need NOT be the same as the name the FSD exports via FS\_NAME, and is NOT the name users employ to refer to the FSD. (They may, however, be the same names.)

Vol Label is the 11-byte ASCII label of the disk/diskette volume. FAT filesystems must ALWAYS use the volume label in the root directory for compatibility reasons. A FSD may use the one in the standard boot sector if it so desires.

The extended BPB structure is a super-set of the conventional BPB structure.

#### 2.0 Functional Characteristics

107

Microsoft Confidential  
OS/2 1.2 IFS Patent Documentation

```

Extended_BPB struct
BytePerSector dw ?
SectorsPerCluster db ? ;See the remark below
ReservedSectors dw ?
NumberOfFATS db ? ;0 for Non Fat system
RootEntries dw ? ;See the remark below
TotalSectors dw ? ;if 0 then use Ext_TotalSectors
MediaDescriptor db ? ;See the remark below
SectorsPerFat dw ? ;See the remark below
SectorsPerTrack dw ?
Heads dw ?
HiddenSectors dd ? ;32 bit hidden sectors
Ext_TotalSectors dd ? ;32 bit total sectors
Extended_BPB ends

```

Any Non Fat-based FSD can ignore, or use the fields of SectorsPerCluster, RootEntries, and SectorsPerFat entries for its own purposes, if necessary.

Although, nothing can prevent a media smaller than 32MB from using the Ext\_TotalSectors, instead of TotalSectors. For compatibility reasons, IBM FORMAT will use TotalSectors for media <= 32MB.

#### 2.1.8.5 IFS Commands

##### 2.1.8.5.1 IFS CONFIG.SYS Function:

##### 2.1.8.5.1.1 IFS:

Purpose The IFS command starts (loads and initializes) an FSD.

Format

IFS = (drive:)(path)name(.ext) (parms)

Where (drive:)(path)name(.ext) specifies the FSD to load and initialize.

parms represents an FSD-defined string of initialization parameters.

#### 2.0 Functional Characteristics

108

00059

EP 0 415 346 A2



### 2.1.8.6 IFS API Function Calls

#### File I/O Function Call Summary

|                    |                                                 |
|--------------------|-------------------------------------------------|
| DosBufReset        | Flush file buffers.                             |
| DosChDir           | Change current directory.                       |
| DosChgFilePtr      | Move the file read/write pointer.               |
| DosClose           | Close file handle.                              |
| DosCopy            | Copy file                                       |
| DosDelete          | Delete file.                                    |
| DosDevIoCtl        | I/O control for devices.                        |
| * DosDupHandle     | Duplicate file handle.                          |
| * DosEditName      | Transform source string using editing string    |
| DosFileIO          | Multi-function call.                            |
| DosFileLocks       | Lock/unlock a range of bytes in an opened file. |
| DosFindClose       | Terminate usage of a directory search handle.   |
| DosFindFirst       | Find first matching file.                       |
| DosFindNext        | Find next matching file.                        |
| DosFindNotifyClose | Terminate usage of a find-notify handle.        |
| DosFindNotifyFirst | Start monitoring directory for changes.         |
| DosFindNotifyNext  | Continue monitoring directory for changes.      |
| DosFsAttach        | Attach a drive or device to an FSD.             |
| DosFsCtl           | File system control                             |
| DosMkDir           | Make subdirectory.                              |
| DosMove            | Move a file or subdirectory.                    |
| DosNewSize         | Change size of a file.                          |
| DosOpen            | Open/create a file.                             |

### 2.0 Functional Characteristics

109

|                  |                                      |
|------------------|--------------------------------------|
| DosQCurDir       | Query current directory.             |
| DosQCurDisk      | Query current default drive.         |
| DosQFHandState   | Query file handle state information. |
| DosQFileInfo     | Query file information.              |
| DosQFileMode     | Query file mode.                     |
| DosQFsAttach     | Query attached FSD information.      |
| DosQFsInfo       | Query file system information.       |
| DosQHandType     | Query handle type.                   |
| * DosQPathInfo   | Query path information.              |
| * DosQSysInfo    | Query system information             |
| DosQVerify       | Query the verify setting.            |
| DosRead          | Read from a file.                    |
| DosReadAsync     | Async Read from a file.              |
| DosRmDir         | Remove subdirectory.                 |
| DosScanEnv       | Scan environment.                    |
| DosSearchPath    | Search path.                         |
| DosSelectDisk    | Select disk.                         |
| DosSetFileInfo   | Set file information.                |
| DosSetFileMode   | Set file mode.                       |
| DosSetFHandState | Set file handle state.               |
| DosSetFsInfo     | Set file system information.         |
| DosSetMaxFH      | Define new maximum file handle.      |
| DosSetPathInfo   | Set path information.                |
| + DosSetVerify   | Set verify setting.                  |
| + DosShutdown    | Shutdown File Systems for Power Off  |
| DosWrite         | Write to a file or device.           |

### 2.0 Functional Characteristics

110

END  
00060 1st FILE

DosWriteAsync Asyno write to a file or device.

#### 2.1.8.6.1 Application File I/O Notes:

File handle values of 0xFFFF do not represent actual file handles but are used through-out the file system interface to indicate specific actions to be taken by the file system. Usage of this 'special file handle' where it is not expected by the file system will result in an error.

Null pointers are defined to be 0x00000000 throughout this specification.

Existing file systems that conform to the Standard Application Program Interface (Standard API) described in this section, may not necessarily support all the described information kept on a file basis. When such is the case, FSDs are required to return to the application a null (zero) value for the unsupported parameter.

An FSD may support version levels of files.

Dates have the following format:

|          |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|----------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Date bit | 5 | 4 | 3 | 2 | 1 | 0 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| mapping  | y | y | y | y | y | y | m | m | m | m | d | d | d | d | d | d |

Where:

mm is the month (1-12)

dd is the day of month (1-31)

yy is number of years since 1980

Times have the following format:

|          |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|----------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Time bit | 5 | 4 | 3 | 2 | 1 | 0 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| mapping  | h | h | h | h | h | m | m | m | m | m | m | x | x | x | x | x |

Where:

hh is the binary number of hours (0-23)

mm is the binary number of minutes (0-59)

xx is the binary number of two-second increments

Some API below may return device-driver/device-manager generated errors. Rather than listing the errors with each API, these error codes are:

- \* ERROR\_WRITE\_PROTECT - the diskette in the drive has write-protection

#### 2.0 Functional Characteristics

enabled.

- \* ERROR\_BAD\_UNIT - there is a breakdown of internal consistency between OS/2's mapping between logical drive and device driver. Internal Error.
- \* ERROR\_NOT\_READY - the device driver detected that the device is not ready.
- \* ERROR\_BAD\_COMMAND - there is a breakdown of internal consistency between OS/2's idea of the capability of a device driver and that device driver.
- \* ERROR\_CRC - the device driver has detected a CRC mismatch.
- \* ERROR\_BAD\_LENGTH - there is a breakdown of internal consistency between OS/2's idea of the length of a request packet and the device driver's idea of that length. Internal Error.
- \* ERROR\_SEEK - the device driver detected an error during a seek operation.
- \* ERROR\_NOT\_DOS\_DISK - the disk is not recognized as being OS/2 manageable.
- \* ERROR\_SECTOR\_NOT\_FOUND - the device is unable to find the specific sector.
- \* ERROR\_OUT\_OF\_PAPER - the device driver has detected that the printer is out-of-paper.
- \* ERROR\_WRITE\_FAULT - other write-specific error.
- \* ERROR\_READ\_FAULT - other read-specific error.
- \* ERROR\_GEN\_FAILURE - other error.

There are also errors defined by and specific to the device driver themselves. These are indicated by either 0xFF or 0xFE in the high byte of the error code.

Error codes listed below are not complete as each FSD may generate errors based upon its own circumstances. The errors below are those generated by the IFS router and the FAT file system.

#### 2.0 Functional Characteristics

00061

EP 0 415 346 A2

## 2.1.8.6.2 Application File I/O Function Calls:

### 2.1.8.6.3 DosBufReset - Commit file's cache buffers:

#### Purpose

Flushes requesting process's cache buffers for the specified file handle or for all file handles attached to that process.

#### Format Calling Sequence:

```
EXTRN DosBufReset:FAR
```

```
PUSH WORD FileHandle ; File handle
CALL DosBufReset
```

**Where** FileHandle is the file handle whose buffers are to be flushed. If FileHandle == 0xFFFF, as above except that all file handles attached to the process are flushed.

**Returns:** IF ERROR (AX not = 0)

AX = Error Code:

- \* ERROR\_INVALID\_HANDLE if the specified file handle is invalid or the handle is attached to a physical device.
- \* Device-driver/device-manager errors listed "-----" on page ---.

**Remarks** The file's directory entry is updated, (as if the file had been closed using DosClose), but the file remains in the open state.

The application should be aware that usage of this function to flush all of the requesting process's cache buffers could have the undesirable effect of requiring the user to mount and unmount a large number of removable volumes.

**Named Pipe Considerations** Performs an operation analogous to the defined one of forcing the buffer cache to disk. For named pipes, DosBufReset blocks the caller until all data written by the caller has been successfully read by the other end. This allows communicating processes to synchronize their dialogs.

## 2.1.8.6.4 DosChDir - Change The Current Directory:

**Purpose** Define the current directory for the requesting process.

**Format** Calling Sequence:

```
EXTRN DosChDir:FAR
```

```
PUSH@ ASCIIZ DirName ; Directory path name
PUSH DWORD 0 ; Reserved (must be zero)
CALL DosChDir
```

**Where** DirName is the ASCIIZ directory path name.

**Returns:** IF ERROR (AX not = 0)

AX = Error Code:

- \* ERROR\_FILENAME\_EXCED\_RANGE - the directory path specified is unacceptable to the FSD managing the volume.
- \* ERROR\_PATH\_NOT\_FOUND - the drive letter is invalid, there are too many ..., there are wildcards in the directory path, or a component of the directory path is not present.
- \* ERROR\_NOT\_ENOUGH\_MEMORY - unable to allocate storage for current directory
- \* ERROR\_DRIVE\_LOCKED - the drive is locked by another process
- \* ERROR\_INVALID\_PARAMETER - the DWORD reserved field is not 0.
- \* Device-driver/device-manager errors listed "-----" on page ---.

**Remarks** If any member of the directory path does not exist, then the directory path is not changed. The current directory is changed only for the requesting process.

DosQSysInfo must be used by an application to determine the maximum path length supported by OS/2. The returned value should be used to dynamically allocate buffers that are to be used to store paths. This will ensure that applications function correctly (wrt path lengths) for future versions of OS/2. The path length includes the drive specifier (i.e. d:), the leading '\', and the trailing NUL.

#### 2.1.0.6.5 DosChgFilePtr - Change File Read Write Pointer:

**Purpose** Moves the read/write pointer according to the method specified.

**Format** Calling Sequence:

EXTRN DosChgFilePtr:FAR

```
PUSH WORD FileHandle ; File handle
PUSH DWORD Distance ; Distance to move in bytes
PUSH WORD MoveType ; Method of moving (0, 1, 2)
PUSH@ DWORD NewPointer ; New Pointer Location
CALL DosChgFilePtr
```

**Where**

FileHandle is the file handle.

Distance is the signed distance (offset) to move in bytes.

MoveType is the method of moving (0, 1, 2).

- \* If value = 0, the pointer is moved Distance bytes (offset) from the beginning of the file.
- \* If value = 1, the pointer is moved to the current location plus offset.
- \* If value = 2, the pointer is moved to the end-of-file plus offset. This method can be used to determine the file's size.

NewPointer is a double word area where the system returns the new pointer location.

**Returns:** IF ERROR (AX not = 0)

AX = Error Code:

- \* ERROR\_INVALID\_HANDLE - the specified handle is not valid in the current process, the handle is attached to a physical device
- \* ERROR\_SEEK\_ON\_DEVICE - the handle is attached to a pipe or character device.
- \* ERROR\_INVALID\_FUNCTION - the parameter MoveType is invalid
- \* ERROR\_NEGATIVE\_SEEK - the resulting position within the file is negative.

- \* `ERROR_DIRECT_ACCESS_HANDLE` - the handle is for a direct open and the resulting position is beyond the physical end of the media.

Remarks The read/write pointer in a file is a signed 32-bit number.  
It is illegal to seek to a negative position in a file.  
It is illegal to seek on a character device or pipe.

#### 2.1.8.6.6 `DosClose` - Close a File Handle:

Purpose Closes the specified file handle.

Format Calling Sequence:

`EXTRN DosClose:FAR`

`PUSH WORD FileHandle ; File Handle`  
`CALL DosClose`

Where `FileHandle` is the handle returned by `DosOpen` or `DosMakePipe`.

Returns: IF ERROR (`AX` not = 0)

`AX` = Error Code:

- \* `ERROR_INVALID_HANDLE` - the specified handle is not open or is attached to a physical device.
- \* `ERROR_FILE_NOT_FOUND` - the directory entry that is thought to contain the name of the file doesn't.
- \* Device-driver/device-manager errors listed "-----" on page ---.

Remarks Closing a file handle causes the file to be closed, the directory to be updated, and all internal buffers for that file to be written to the media.

Named Pipe Considerations Closes a pipe by handle. When all handles referencing one end of a pipe are closed the pipe is broken. If the client end closes, no other process can reopen the pipe until the serving end issues a `DosDisconnectNmPipe` followed by a `DosConnectNmPipe`. If the server end closes, the pipe will be deallocated when the last client handle is closed or immediately if the pipe is already broken.

`DosQSysInfo` must be used by an application to determine the maximum path length supported by OS/2. The returned value should be used to dynamically allocate buffers that are to be used to store paths. This will ensure that applications function correctly (wrt path lengths) for future versions of OS/2. The path length includes the drive specifier (i.e. d:), the leading '\', and the trailing NUL.

# 2.1.8.6.7 DosCopy - Copy file:

**Purpose** Copies the specified file or subdirectory to target file or subdirectory

**Format** Calling Sequence:

EXTRN DosCopy:FAR

PUSH# ASCIIZ SourceName ; Source path name  
PUSH# ASCIIZ TargetName ; Target path name  
PUSH WORD OpMode ; Operation mode  
PUSH DWORD 0 ; Reserved (must be zero)  
CALL DosCopy

**Where** SourceName is the ASCIIZ path name of the source file, subdirectory, or character device. Wildcards are not allowed.

TargetName is the ASCIIZ path name of the target file, subdirectory, or character device. Wildcards are not allowed.

OpMode is a word-length bit map that defines how the DosCopy function will be done.

The bit field mapping is shown as follows:

OpMode 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0  
bits R R R R R R R R R R R R R R R A E

**E** Existing Target File Disposition

If E = 0; Do not copy the source file to the target if the file name already exists within the target directory. If a single file is being copied and the target already exists, an error is returned.

If E = 1; Copy the source file to the target even if the file name already exists within the target directory.

**A** Append the source file to the target file's end of data.

If A = 0; Replace the target file with the source file.

If A = 1; Append the source file to the target file's end of data. This is ignored when copying a directory or if the target file doesn't exist.

**R** Reserved and must be set to zero.

**Returns:** IF ERROR (AX not = 0)

\* AX = Error Code:

- ERROR\_FILENAME\_EXCED\_RANGE - the path specified is unacceptable to the FSD managing the volume.
- ERROR\_PATH\_NOT\_FOUND - Either the old or new path can't be found.
- ERROR\_SHARING\_VIOLATION - One of the files is open.
- ERROR\_INSUFFICIENT\_DISK\_SPACE
- ERROR\_ACCESS\_DENIED - Target of a file copy is read-only or target of a file copy already exists and E = 0.
- ERROR\_DIRECTORY - Cannot be copied to itself. Cannot be copied to a file. Cannot be copied to one of its subdirectories.
- ERROR\_FILE\_NOT\_FOUND
- ERROR\_NOT\_DOS\_DISK
- ERROR\_DRIVE\_LOCKED
- ERROR\_SHARING\_BUFFER\_EXCEEDED
- ERROR\_INVALID\_PARAMETER

**Remarks** The source and target may be on different drives.

Wildcard characters are not allowed in the source or target name.

When a subdirectory is specified to be copied and an I/O error occurs, the specific file being copied from the source to the target at the time of the error will be deleted from the target path and the DosCopy function terminated.

When a file (non-directory) is specified to be copied, in the event of an I/O error, the file will be deleted from the target path in replace mode, or resized to its original size in append mode, and the DosCopy function terminated.

All files and/or directories (as applicable) in the source path are added to the target path.

Read-only files in the target path are not replaced. When copying

a directory with E = 0, they are ignored; when copying a directory with E = 1 or when copying a single file, an error is returned.

When TargetName indicates a device name, SourceName must indicate a file, not a directory.

When copying a single file with A = 1 (Append mode), the operation will proceed even if the file exists and E = 0. In other words, the E bit is only significant when replacing, not when appending.

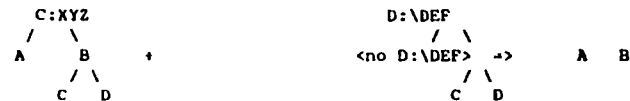
OpMode bit flags A and E are undefined (ignored) when the target is a device.

DosCopy will copy the source's attributes (date of creation, time of creation, ...) to the target.

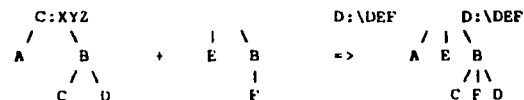
DosCopy will copy the source's EAs to the target when creating a new subdirectory, when creating a new file, or when replacing an existing file. It will not copy the source's EAs to an existing file when appending to it or to an existing directory when copying files to that directory. If the file system of the destination does not support EAs, DosCopy will terminate the operation and return the error.

DosCopy takes the source subdirectory structure (one node) and CREATE/APPEND a similar directory structure to a destination node:

DosCopy C:\XYZ D:\DEF



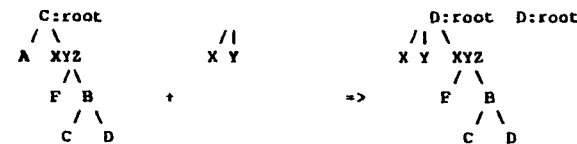
- or -



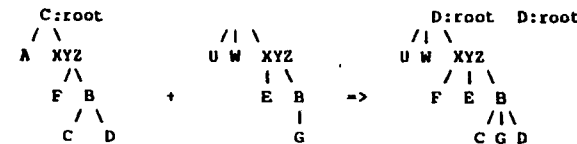
Subdirectories from the source path "C:\XYZ\B" of the second example, are appended (or logically added) to like-named subdirectories in the target path "D:\DEF\B".

A second example of DosCopy operation:

DosCopy C:\XYZ D:\XYZ



- or -



#### 2.1.8.6.8 DosDelete - Delete a File:

**Purpose** Removes a directory entry associated with a filename.

**Format** Calling Sequence:

EXTRN DosDelete:FAR

```
PUSH# ASCIIZ FileName ; Filename path
PUSH DWORD 0 ; Reserved (must be zero)
CALL DosDelete
```

**Where** FileName is the ASCIIZ file name.

**Returns:** IF ERROR (AX not - 0)

AX - Error Code:

- ERROR\_FILENAME\_EXCED\_RANGE - the path specified is unacceptable to the FSD managing the volume.
- ERROR\_PATH\_NOT\_FOUND - the drive letter is invalid, there are too many ..., there are wildcards present, or a component of the directory path is not present.
- ERROR\_FILE\_NOT\_FOUND - the filename at the end of the path is not found.
- ERROR\_ACCESS\_DENIED - the path specified a directory or a device, or the file is read-only.
- ERROR\_SHARING\_VIOLATION - the file specified is currently in use.
- ERROR\_SHARING\_BUFFER\_EXCEEDED - there is not enough memory to hold sharing information.
- Device-driver/device-manager errors listed "-----" on page ---.

**Remarks** Wildcard characters are not allowed in any part of the ASCIIZ string.

Read-only files cannot be deleted by this call. To delete a read-only file, you must first use function DosSetFileMode (Set File Mode) to change the file's read-only attribute to 0, then delete the file.

DosQSysInfo must be used by an application to determine the

maximum path length supported by OS/2. The returned value should be used to dynamically allocate buffers that are to be used to store paths. This will ensure that applications function correctly (wrt path lengths) for future versions of OS/2. The path length includes the drive specifier (i.e. d:), the leading '\', and the trailing NUL.



#### 2.1.8.6.9 DosDevIOctl - I/O Control for Devices:

**Purpose** Perform control functions on the device specified by the file or device handle.

**Format** Calling Sequence:

EXTRN DosDevIOctl:FAR

```
PUSH@ OTHER Data ; Data area
PUSH@ OTHER ParmList ; Command arguments
PUSH WORD Function ; Device function
PUSH WORD Category ; Device category
PUSH WORD DevHandle ; Specifies the device
CALL DosDevIOctl
```

EXTRN DosDevIOctl2:FAR

```
PUSH@ OTHER Data ; Data area
PUSH WORD DataLength ; Data area length
PUSH@ OTHER ParmList ; Command arguments
PUSH WORD ParmListLength ; Command arg's list length
PUSH WORD Function ; Device function
PUSH WORD Category ; Device category
PUSH WORD DevHandle ; Specifies the device
CALL DosDevIOctl2
```

**Where** Data is a data area.

DataLength is the length of the data buffer.

ParmList is a command-specific argument list.

ParmListLength is the length of the command-specific argument list.

Function is the device-specific function code. The valid range is 0 to 255.

Category is the device category. The valid range is 0 to 255.

DevHandle is a device handle returned by DosOpen.

**Returns:** IF ERROR (AX not = 0)

AX - Error Code:

- ERROR\_INVALID\_HANDLE - the handle specified is either not in use, is attached to a physical device and category 9 is not specified or is not attached to a physical device and category 9 is specified.

- ERROR\_INVALID\_FUNCTION - the specified function is illegal for the particular category and handle.
- ERROR\_INVALID\_CATEGORY - the specified category is illegal for the particular function and handle.
- ERROR\_INVALID\_DRIVE - the drive specified in a set-logical-drive-map call is not supported by the device driver.
- ERROR\_INVALID\_PARAMETER - (DosDevIOctl2) a 0 length was specified with a non-null pointer for either Data or ParmList.
- Device-driver/device-manager errors listed "-----" on page ---.

**Remarks** This function provides a generic, expandable IOCTL facility.

A null (zero) value for Data specifies that this parameter is not defined for the generic IOCTL function being specified. A null value for Data causes the value passed in DataLength to be ignored.

A null (zero) value for ParmList specifies that this parameter is not defined for the generic IOCTL function being specified. A null value for ParmList causes the value passed in ParmListLength to be ignored.

The kernel will format a generic IOCTL packet and call the device driver. Since v1.0 and v1.1 device drivers do not understand generic IOCTL packets with DataLength and ParmListLength, the kernel will not pass these fields to the device driver. Device drivers that are marked as being level 2 or higher (see device driver header spec) must support receipt of the generic IOCTL packets with associated length fields.

It is illegal to pass in a non-null pointer with a zero length.

2.1.8.6.10 DosDupHandle - Duplicate a File Handle:

**Purpose** Returns a new file handle for an open file that refers to the same file at the same position.

**Format** Calling Sequence:

EXTRN DosDupHandle:FAR

PUSH WORD OldFileHandle ; Existing file handle  
PUSH@ WORD NewFileHandle ; New file handle  
CALL DosDupHandle

**Where** OldFileHandle is the current file handle.

NewFileHandle is a word file handle that is made the duplicate of OldFileHandle or is 0xFFFF. A value of 0xFFFF causes the DOS to allocate a new file handle and use that as the destination of the duplication. Otherwise, the function uses NewFileHandle as the destination of the duplication.

**Returns:** IF ERROR (AX not = 0)

AX = Error Code:

- \* ERROR\_INVALID\_HANDLE - the OldFileHandle is invalid or is attached to a physical device.
- \* ERROR\_TOO\_MANY\_OPEN\_FILES - NewFileHandle is 0xFFFF and all handles for the process are in use.
- \* ERROR\_INVALID\_TARGET\_HANDLE - the specified NewFileHandle is beyond the range allocated to the process, either by default, or by DOSSETMAXFH.
- \* Device-driver/device-manager errors listed "-----  
-----" on page ---.

**Remarks** Duplicating the handle duplicates and ties all handle-specific information between OldFileHandle and NewFileHandle.

A file handle value other than 0xFFFF specified in NewFileHandle causes that file handle to be closed before it is redefined as the duplicate of OldFileHandle.

The valid values for NewFileHandle include: 0xFFFF, 0x0000 (Standard Input), 0x0001 (Standard Output), 0x0002 (Standard Error), or the handle of any currently open file. Avoid using other arbitrary values.

If you move the read/write pointer of either handle by a DosRead, DosWrite, or DosChgFilePtr function call, the pointer for the other handle is also changed.

Issuing a DosClose against a file handle does not affect the duplicate handle.

\* 2.1.8.6.11 DosEditName - Transform source string using editing string:

\* Purpose Transform a source string into a destination string using an editing string and OS/2 meta editing semantics.

\* Format Calling Sequence:

```

EXTRN DosEditName:FAR

PUSH WORD EditLevel ; Level of meta editing semantics
PUSH# ASCIIZ SourceString ; String to transform
PUSH# ASCIIZ EditString ; Editing string
PUSH# OTHER TargetBuf ; Destination string buffer
PUSH WORD TargetBufLen ; Destination string buffer length
CALL DosEditName

```

\* Where EditLevel is the version of meta editing semantics to use in transforming the source string.

\* If EditLevel value is 0x0001, then OS/2 version 1.2 editing semantics are used.

\* SourceString is the ASCIIZ string to transform. It should contain just the component of the pathname to edit, not the entire path.

\* EditString is the ASCIIZ string to use for editing.

\* TargetBuf is the buffer to store the resulting ASCIIZ string in.

\* TargetBufLen is the length of the buffer to store the resulting string in.

\* Returns: IF ERROR (AX not = 0)

\* AX = Error Code:

- \* ERROR\_INVALID\_PARAMETER - the EditLevel is invalid.
- \* ERROR\_INVALID\_NAME - the SourceString or EditString contains a drive letter or path characters.

\* Remarks Typical uses of this API are copy and rename/move. For a SourceString "foo.bar" and an EditString ".baz", the destination string is "foo.baz" for OS/2 version 1.2 meta editing semantics.

The destination string will be uppercased.

+ 2.1.8.6.12 DosEnumAttribute - Enumerate the extended attributes:

+ Purpose Enumerates information for a specific file or subdirectory.

+ Format Calling Sequence:

```

EXTRN DosEnumAttribute:FAR

PUSH WORD RefType ; Type of reference
PUSH# OTHER FileRef ; Handle or Name
PUSH DWORD EntryNum ; Starting entry in EA list
PUSH# OTHER EnumBuf ; Data buffer
PUSH DWORD EnumBufSize ; Data buffer size
PUSH# DWORD EnumCnt ; Count of entries to return
PUSH DWORD InfoLevel ; Level of information requested
PUSH DWORD 0 ; Reserved (must be zero)
CALL DosEnumAttribute

```

+ Where RefType indicates whether FileRef points to a handle or an ASCIIZ name. Reftype = 0 indicates a handle; Reftype = 1 indicates that FileRef points to an ASCIIZ name of a file or directory.

+ FileRef is a pointer to a WORD handle that was obtained >from DOSOPEN/DOSOPEN2, or to an ASCIIZ name of a file or directory.

+ EntryNum is the ordinal of the entry in the EA list from which to start copying information into EnumBuf.

+ The EntryNum value of zero is reserved, and the first EA is indicated by a value of 1.

+ EnumBuf is a pointer to the buffer where the requested information is returned.

+ EnumBufSize is the size of EnumBuf.

+ EnumCnt is, on input, the number of EAs whose information has been requested. The system will change this to the number actually returned. An EnumCnt of -1 is treated specially in that it will return as many entries as will fit in the buffer specified.

+ InfoLevel is the level of information required.

+ Level 0x00000001 information is returned in the following format:

```
+ struct {
+ unsigned char reserved; /* must be 0
+ unsigned char cbName; /* length of name excluding NULL
+ unsigned short cbValue; /* length of value
+ unsigned char szName[]; /* asciiz name
+ }
```

+ The fields in the structure are on a one-to-one mapping for the fields in an FEA structure.

+ The information for the next EA will be stored adjacent to the previous one. An application that wishes to continue on from one DosEnumAttribute call would have to set EntryNum to the previous value plus the returned value in EnumCnt.

+ The size of buffer needed to hold an EA can be obtained from the enumerated information. An EA would occupy:

```
+ one byte (for fea_Reserved) +
+ one byte (for fea_cbName) +
+ two bytes (for fea_cbValue) +
+ value of cbName (for the name of the EA) +
+ one byte (for terminating NULL in fea_cbName) +
+ value of cbValue (for the value of the EA)
```

+ Returns: IF ERROR (AX not = 0)

+ AX = Error Code:

- + \* ERROR\_FILENAME\_EXCED\_RANGE - the path specified is unacceptable to the FSD managing the volume.
- + \* ERROR\_INVALID\_HANDLE - the handle is not in use or is attached to a physical device.
- + \* ERROR\_ACCESS\_DENIED - the handle is opened as write-only.
- + \* ERROR\_PATH\_NOT\_FOUND - a component of FileRef is not found.
- + \* ERROR\_NOT\_ENOUGH\_MEMORY - the specified buffer is too short for the returned data.
- + \* ERROR\_INVALID\_LEVEL - the specified InfoLevel is not supported.
- + \* ERROR\_INVALID\_PARAMETER - an invalid parameter was specified.

- + \* ERROR\_BUFFER\_OVERFLOW - the buffer is not big enough to hold data for even one Extended Attribute.
- + \* Device-driver/device-manager errors listed "-----" on page ----.

+ Remarks It is important to note that the use of EntryNum in no way guarantees the specific ordering of the EAs. It is only a tool to enumerate the EAs, and should not be used to "back up" to the nth EA, since the EAs are subject to change by other tasks. So, for example, the EA returned when EntryNum is 11 may not necessarily be the EA returned when EntryNum is 11 for a subsequent call if another task had performed a SET operation on the EAList.

+ No explicit EA sharing is done for DosEnumAttribute. Implicit sharing exists if the caller passes in the handle of an open file, since sharing access to the associated file is required to modify its EAs. No sharing is done if the caller passes in the pathname. This means that if some other process is editing the EAs, and changes them between two calls to DosEnumAttribute, inconsistent results may be returned (i.e. see same value twice, miss seeing values, etc.) To prevent the modification of EAs for the handle case, make sure that the file is open in deny-write mode. To prevent the modification of EAs for the pathname case, open the file in deny-write mode before calling DosEnumAttribute. For the directory name case, no sharing is possible.

+ It should be noted that for RefType = 1, the EAs returned by this API are current only when the call was made, and they may have been changed by another thread or process in the meantime.

#### 2.1.8.6.13 DosFileIO - File I/O:

Purpose Perform multiple lock, unlock, seek, read, and write I/O.

Format Calling Sequence:

EXTRN DosFileIO:FAR

```
PUSH WORD FileHandle ; File handle
PUSH# OTHER CommandList ; Ptr to command buffer
PUSH WORD CommandListLen ; Length of command buffer
PUSH# WORD ErrorOffset ; Ptr to command error offset
CALL DosFileIO
```

Where FileHandle is the handle of the file of interest.

CommandList is a list that contains entries indicating what commands will be performed. The operations (CmdLock, CmdUnlock, CmdSeek, CmdIO) are performed as atomic operations until all are complete or until one fails. CmdLock allows a multiple range lock to be executed as an atomic operation. Each operation will be guaranteed to execute in one piece. Unlike CmdLock, CmdUnlock cannot fail as long as the parameters to it are correct, and the calling application had done a Lock earlier, so it can be viewed as atomic.

For CmdLock, the command format is:

```
CmdLock Struc
Cmd dw 0 ; 0 for lock operations
LockCnt dw ? ; Number of locks that follow
Timeout dd ? ; ms timeout for success of all locks
CmdLock Ends
```

which is followed by a series of records of the following format:

```
Lock Struc
Share dw ? ; 0 for exclusive, 1 for read-only access
Start dd ? ; start of locked region
Length dd ? ; length of locked region
Lock Ends
```

If a lock within a CmdLock causes a timeout, none of the other locks within the scope of CmdLock are in force since the lock operation is viewed as atomic.

CmdLock.Timeout is the count in milliseconds, until the requesting process is to resume execution if the requested locks are not available. The meaning of the values specified are:

- \* 0xFFFFFFFF means the process will wait indefinitely until the requested locks become available.

- \* 0x00000000 means return immediately to the requesting process
- \* Otherwise, CmdLock.Timeout is the number of milliseconds to wait until the requested locks become available.

Lock.Share defines the type of access other processes may have to the file-range being locked.

- \* If Lock.Share == 0, other processes have 'No-Access' to the locked range. The current process will have both read and write access to the locked range. No region with Lock.Share == 0 may overlap with any other locked region.

- \* If Lock.Share == 1, other processes have 'Read-Only' access to the locked range. The current process will have 'Read-Only' access to the locked range also. A range locked with Lock.Share == 1 may overlap with other ranges locked with Lock.Share == 1, but may not overlap with other ranges locked with Lock.Share == 0.

For CmdUnlock, the command format is:

```
CmdUnlock Struc
Cmd dw 1 ; 1 for unlock operations
UnlockCnt dw ? ; Number of unlocks that follow
CmdUnlock Ends
```

which is followed by a series of records of the following format:

```
Unlock Struc
Start dd ? ; start of locked region
Length dd ? ; length of locked region
Unlock Ends
```

For CmdSeek, the command format is:

```
CmdSeek Struc
Cmd dw ? ; 2 for seek operation
Method dw ? ; 0 for absolute,
; 1 for relative to current,
; 2 for relative to EOF.
Position dd ? ; file seek position or delta
Actual dd ? ; actual position seeked to
CmdSeek Ends
```

For CmdIO, the command format is:

```
CmdIO Struc
Cmd dw ? ; 3 for read, 4 for write
Buffer# dd ? ; ptr to the data buffer
```

Microsoft Confidential  
OS/2 1.2 IFS Patent Documentation

BufferLen dw ? ; number of bytes requested  
Actual dw ? ; number of bytes actually  
; transferred  
CmdIO Ends

CommandListLen is the length in bytes of CommandList.

ErrorOffset points to where the system stores the byte offset relative to the start of the record of where an error occurred.

Returns: IF ERROR (AX not = 0)

AX = Error Code:

- \* ERROR\_INVALID\_PARAMETER - invalid command
- \* ERROR\_ACCESS\_DENIED - the file is not accessible because another process has access to it.
- \* ERROR\_SHARING\_BUFFER\_EXCEEDED - there is not enough memory to hold sharing information.
- \* ERROR\_INTERRUPT - the current thread received a signal.
- \* ERROR\_SEEK\_ON\_DEVICE - the handle is attached to a pipe or character device.
- \* ERROR\_NEGATIVE\_SEEK - the resulting position within the file is negative.
- \* ERROR\_DIRECT\_ACCESS\_HANDLE - the specified handle is opened with the direct open bit set.
- \* ERROR\_INVALID\_HANDLE - the file handle specified is not in use or is attached to a physical device.
- \* ERROR\_LOCK\_VIOLATION - a lock operation timed out or a read/write operation encountered a lock.
- \* Device-driver/device-manager errors listed "-----" on page ---.

Remarks This function provides a simple mechanism for combining the following operations into a single request and providing improved performance particularly in a networking environment:

- \* unlocking and locking of multiple file ranges
- \* changing of the file position pointer
- \* read and/or write IO

Microsoft Confidential  
OS/2 1.2 IFS Patent Documentation

Provides a simple mechanism for excluding other processes read/write or write access to regions of the file. If another process attempts to read or write a 'No-access' region, or attempts to write in a 'Read-only' region, the system call will return an error indicating exclusion to that region. If the time-out occurs before the locking can be completed, the function returns to the caller with an unsuccessful return code.

The caller may return after the timeout period has expired without receiving an ERROR\_SEM\_TIMEOUT. Therefore, semaphore timeout values should not be used for exact timing and sequencing.

A range to be locked must first be cleared of any locked subranges or locked overlapping ranges.

#### 2.1.8.6.14 DosFileLocks - File Lock Manager:

**Purpose** Unlock and/or lock a range in an opened file.

**Format** Calling Sequence:

EXTRN DosFileLocks:FAR

```
PUSH WORD FileHandle ; File handle
PUSH# OTHER UnlockRange ; Unlock range
PUSH# OTHER LockRange ; lock range
CALL DosFileLocks
```

**Where** FileHandle is the file handle.

UnlockRange identifies the range in the file of the region to be unlocked:

```
UnlockRange struc
FileOffset dd ? ; offset where unlock begins
RangeLength dd ? ; length of region unlocked
UnlockRange ends
```

A null pointer (a dword of zero) to UnlockRange specifies that unlocking is not required.

LockRange identifies the range in the file of the region to be locked:

```
LockRange struc
FileOffset dd ? ; offset where lock begins
RangeLength dd ? ; length of region locked
LockRange ends
```

A null pointer (a dword of zero) to LockRange specifies that locking is not required.

**Returns:** IF ERROR (AX not = 0)

AX = Error Code:

- \* ERROR\_INVALID\_HANDLE - handle specified is not open or is attached to a physical device.
- \* ERROR\_LOCK\_VIOLATION - the range specified for locking overlapped a previously locked range or the range specified for unlocking did not exactly match a previously locked range.
- \* ERROR\_SHARING\_BUFFER\_EXCEEDED - there is no more room for locks.

**Remarks** This function provides a simple mechanism for unlocking and/or

locking a file range.

If unlocking is specified, the function first unlocks the specified area using UnlockRange. After UnlockRange is processed, then the locking of a range (if specified via LockRange), is done.

Closing a file with locks still in force causes the locks to be released in no defined order.

Terminating with a file open and having issued locks on that file causes the file to be closed and the locks to be released in no defined order.

Provides a simple mechanism for excluding other processes read/write access to regions of the file. The locked regions can be anywhere in the logical file. Locking beyond end-of-file is not an error. It is expected that the time in which regions are locked will be short. Duplicating the handle duplicates access to the locked regions. Access to the locked regions is not duplicated across the DosExecPgm system call. The proper method for using locks is not to rely on being denied read or write access, but attempting to lock the region desired and examining the error code.

A range to be locked must first be cleared of any locked subranges or locked overlapping ranges.

#### 2.1.8.6.15 DosFindClose - Close Find Handle:

**Purpose** Closes the association between a directory handle and a DosFindFirst or DosFindNext directory search function.

**Format** Calling Sequence:

```
EXTRN DosFindClose:FAR
```

```
PUSH WORD DirHandle ; Directory search handle
CALL DosFindClose
```

**Where**

DirHandle is the handle previously associated by the system with a DosFindFirst or DosFindNext directory search function.

**Returns:** IF ERROR (AX not = 0)

AX = Error Code:

\* ERROR\_INVALID\_HANDLE - the specified DirHandle is invalid.

**Remarks** When DosFindClose is issued, a subsequent DosFindNext for the closed DirHandle will fail unless an intervening DosFindFirst has been issued specifying DirHandle.

1 DosQSysInfo must be used by an application to determine the  
 1 maximum path length supported by OS/2. The returned value should  
 1 be used to dynamically allocate buffers that are to be used to  
 1 store paths. This will ensure that applications function  
 1 correctly (wrt path lengths) for future versions of OS/2. The  
 1 path length includes the drive specifier (i.e. d:), the leading  
 1 '\ ' and the trailing NUL.

#### 2.1.8.6.16 DosFindFirst - Find First Matching File:

**Purpose** Finds the first filename that matches the specified file specification.

**Format** Calling Sequence:

```
EXTRN DosFindFirst:FAR
```

```
PUSH@ ASCIIZ FileName ; File path name
PUSH@ WORD DirHandle ; Directory search handle
PUSH WORD Attribute ; Search attribute
PUSH@ OTHER ResultBuf ; Result buffer
PUSH WORD ResultBufLen ; Result buffer length
PUSH@ WORD SearchCount ; # of entries to find
PUSH DWORD 0 ; Reserved (must be zero)
CALL DosFindFirst
```

```
EXTRN DosFindFirst2:FAR
```

```
PUSH@ ASCIIZ FileName ; File path name
PUSH@ WORD DirHandle ; Directory search handle
PUSH WORD Attribute ; Search attribute
PUSH@ OTHER ResultBuf ; Result buffer
PUSH WORD ResultBufLen ; Result buffer length
PUSH@ WORD SearchCount ; # of entries to find
PUSH WORD FileInfoLevel ; File data required
PUSH DWORD 0 ; Reserved (must be zero)
CALL DosFindFirst2
```

**Where** FileName is the ASCIIZ path name of the file to be found.

DirHandle is the directory handle associated by the DOS with this specific request. A DirHandle value of 0x0001 is defined to be always available. A DirHandle value of 0xFFFF indicates to allocate a handle to the user. The handle is returned by overwriting the 0xFFFF. Reuse of this DirHandle in another DosFindFirst closes the association with the previously related DosFindFirst and opens a new association with the current DosFindFirst.

FileInfoLevel is the level of file information required. DOSFINDFIRST (and DOSFINDNEXT on handles returned by DOSFINDFIRST) always returns level 0x0001 information.

Level 'n' file information is returned in the format described in ResultBuf.

\* If FileInfoLevel value is 0x0001, then if Attribute is set for



hidden, system files, or directory files, it is considered as an inclusive search. All normal file entries plus all entries matching the specified attributes are returned. To look at all directory entries except the volume label, Attribute may be set to hidden + system + directory (all 3 bits on).

Attribute is the attribute used in searching for the file. Attribute cannot specify the volume label. Volume labels may be queried using DosQFInfo.

ResultBuf is where the FSD returns the results of the qualified directory search. The information returned is guaranteed to be at least as accurate as the most recent DosClose or DosBufReset.

For level 0x0001, directory information (find record) is returned in the following format:

```
struct {
 unsigned short dateCreate;
 unsigned short timeCreate;
 unsigned short dateAccess;
 unsigned short timeAccess;
 unsigned short dateWrite;
 unsigned short timeWrite;
 long cbEOF;
 long cbAlloc;
 unsigned short attr;
 unsigned char cbName;
 unsigned char szName[];
};
```

For level 0x0002, the cbList field is added to the level 0x0001 structure, and returned in the following format:

```
struct {
 unsigned short dateCreate;
 unsigned short timeCreate;
 unsigned short dateAccess;
 unsigned short timeAccess;
 unsigned short dateWrite;
 unsigned short timeWrite;
 long cbEOF;
 long cbAlloc;
 unsigned short attr;
 unsigned long cbList;
 unsigned char cbName;
 unsigned char szName[];
};
```

The cbList field can be used to calculate the size of the buffer necessary for level 0x0003.

For level 0x0003, ResultBuf is an EAOP structure on input. fpGEAList points to a GEA list defining the attribute names whose values will be returned. fpFEAList is ignored. In case of error, offError will point to the offending GEA entry.

On output, ResultBuf will contain the EAOP structure (unchanged) followed by a packed set of records that consist of the attributes record described in level 0x0001, excluding the cbName and szName fields, followed by an FEAList structure followed by a byte length of the name of the object followed by the asciiz name of the object matched by the input pattern.

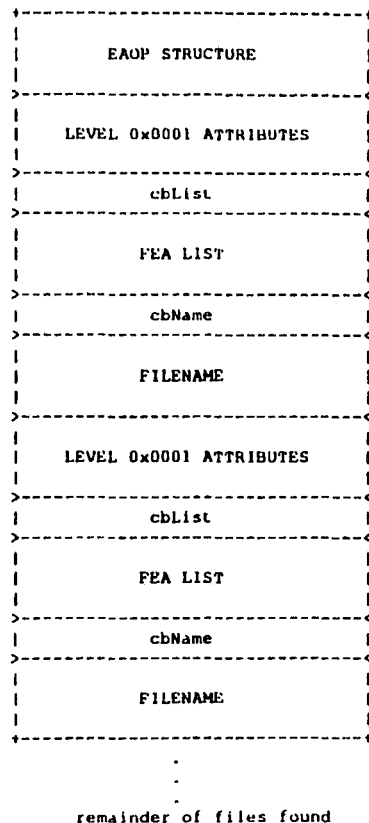


Figure 9. DOSFINDFIRST return buffer format

Even if there is not enough room to hold all the requested information, (ie ERROR\_BUFFER\_OVERFLOW) the cbList field of the FEA list will still be valid, assuming there's at least enough

space to hold it. For the buffer overflow case, cbList will contain the size of the entire EA set for the file, even if only a subset of its attributes were requested.

If a particular attribute is not found attached to the object, there will be an FEA structure containing the name of the attribute but with a zero-length value.

ResultBufLen is the length of ResultBuf.

SearchCount is the number of matching entries requested in ResultBuf. The file system uses this field to store the number of entries actually returned.

Returns: IF ERROR (AX not = 0)

AX = Error Code:

- \* ERROR\_INVALID\_PARAMETER - the search attribute is invalid or the specified search count is 0.
- \* ERROR\_INVALID\_HANDLE - the handle specified is never allocated.
- \* ERROR\_FILENAME\_EXCED\_RANGE - the path specified is unacceptable to the FSD managing the volume.
- \* ERROR\_META\_EXPANSION\_TOO\_LONG - the FAT FS is unable to expand meta characters correctly.
- \* ERROR\_PATH\_NOT\_FOUND - the drive letter is invalid, there are too many ..., there are wildcards present before the last component, or a component of the directory path is not present.
- \* ERROR\_BUFFER\_OVERFLOW - the specified buffer is too small to support a single entry.
- \* ERROR\_NO\_MORE\_SEARCH\_HANDLES - there are too many search handles in use for the current process.
- \* ERROR\_NO\_MORE\_FILES - there are no matching files
- \* Device-driver/device-manager errors listed "-----" on page ----.
- \* ERROR\_INVALID\_EA\_NAME - there is an illegal character in an EA name. The offError field points to the offending GEA.
- \* ERROR\_EA\_LIST\_INCONSISTENT - the cbList does not match the sum of the lengths of the GEA structures.

- \* ERROR\_BUFFER\_OVERFLOW - The buffer was not big enough to hold the information, excluding the EAs, for even one entry that was found.
- \* ERROR\_EAS\_DIDNT\_FIT - The buffer was not big enough to hold the EAs for the first matching entry being returned in the buffer. This occurs for infolevel 3 where the first matching entry cannot be returned together with its EAs because the EAs don't fit in the buffer.

Remarks DosFindNext uses the directory handle to repeat the related DosFindFirst.

The filename in FileName can contain wildcard characters.

\* Any non-zero return code except ERROR\_EAS\_DIDNT\_FIT indicates no handle has been allocated. This includes such "non-error" indicators as ERROR\_NO\_MORE\_FILES.

\* In the case of ERROR\_EAS\_DIDNT\_FIT, a search handle is returned, and a subsequent call to DosFindNext will get the next matching entry in the directory. You may use DosQPathInfo to retrieve the EAs for the matching entry by using the EA arguments that were used for the FindFirst2 call and the name that was returned by FindFirst2.

\* In the case of ERROR\_EAS\_DIDNT\_FIT, only information for the first matching entry is returned. This entry is the one whose EAs did not fit in the buffer. The information returned is in the format of that returned for infolevel 2. No further entries are returned in the buffer even if they could fit in the remaining space.

#### 2.1.8.6.17 DosFindNext - Find Next Matching File:

Purpose Finds the next directory entry matching the name that is specified on the previous DosFindFirst or DosFindNext function call.

Format Calling Sequence:

EXTRN DosFindNext:FAR

```
PUSH WORD DirHandle ; Directory handle
PUSH@ OTHER ResultBuf ; Result buffer
PUSH WORD ResultBufLen ; Result buffer length
PUSH@ WORD SearchCount ; # of entries to find
CALL DosFindNext
```

Where DirHandle is the handle (previously returned by DOS) associated with a previous DosFindFirst or DosFindNext function call.

ResultBuf is where the FSD returns the results of the qualified directory search. The information returned is guaranteed to be at least as accurate as the most recent DosClose or DosBufReset.

For the continuation of an infolevel 3 search, this buffer should contain input in the same format as a DosFindFirst2 infolevel 3 search.

ResultBufLen is the length of ResultBuf.

SearchCount is the number of matching entries requested in ResultBuf. The file system uses this field to store the number of entries actually returned.

Returns: IF ERROR (AX not = 0)

AX = Error Code:

- \* ERROR\_INVALID\_PARAMETER - a search count of zero is specified.
- \* ERROR\_INVALID\_HANDLE - the specified search handle is not in use.
- \* ERROR\_BUFFER\_OVERFLOW - the specified buffer was not big enough to hold the information, excluding the EAs, for even one entry that was found.
- \* ERROR\_EAS\_DIDNT\_FIT - The buffer was not big enough to hold the EAs for the first matching entry being returned in the buffer. This occurs for infolevel 3 where the first matching entry cannot be returned together with its EAs because the EAs don't fit in the buffer.

00078

- \* ERROR\_NO\_MORE\_FILES - there are no more files matching files.
- \* Device-driver/device-manager errors listed "-----  
-----" on page ---.

Remarks If no more matching files are found, an error code is returned.

+ If an ERROR\_BUFFER\_OVERFLOW error is returned, the further calls  
+ to DosFindNext will start the search from the same entry.

+ If an ERROR\_EAS\_DIDNT\_FIT error is returned, then the buffer was  
+ too small to hold the EAs for the first matching entry being  
+ returned. A subsequent call to FindNext will get the next matching  
+ entry. This enables the search to continue if the EAs being  
+ returned are too big to fit in the buffer. You may use  
+ DosQPathInfo to retrieve the EAs for the matching entry by using  
+ the EA arguments that were used for the FindFirst2 call and the  
+ name that was returned by FindFirst2.

+ In the case of ERROR\_EAS\_DIDNT\_FIT, only information for the first  
+ matching entry is returned. This entry is the one whose EAs did  
+ not fit in the buffer. The information returned is in the format  
+ of that returned for Infolevel 2. No further entries are returned  
+ in the buffer even if they could fit in the remaining space.

#### 2.1.8.6.18 DosFindNotifyClose - Close Find-Notify Handle:

Purpose Closes the association between a 'Find-Notify' handle and a  
DosFindNotifyFirst or DosFindNotifyNext function.

Format Calling Sequence:

EXTRN DosFindNotifyClose:FAR

PUSH WORD DirHandle ; Find-notify handle  
CALL DosFindNotifyClose

Where

DirHandle is the handle previously associated by the system with  
a DosFindNotifyFirst or DosFindNotifyNext function.

Returns: IF ERROR (AX not = 0)

AX = Error Code:

- \* ERROR\_INVALID\_HANDLE - the specified handle is not in use.

Remarks When DosFindNotifyClose is issued, a subsequent DosFindNotifyNext  
for the closed DirHandle will fail unless an intervening  
DosFindNotifyFirst has been issued specifying DirHandle.

+ If a DosFindNotifyClose is issued on a DirHandle while a thread  
+ that issued a DosFindNotifyNext for the same DirHandle is waiting  
+ for the requested changes or the timeout, the close operation will  
+ abort the DosFindNotifyNext and will succeed. The  
+ DosFindNotifyNext thread will return immediately with an  
+ ERROR\_FINDNOTIFY\_HANDLE\_CLOSED return code. If the  
+ DosFindNotifyNext call has timed-out or counted the requested  
+ number of changes but has not returned, the DosFindNotifyClose  
+ thread will sleep until the next operation completed.

#### 2.1.8.6.19 DosFindNotifyFirst - Start monitoring directory for changes:

Purpose Start monitoring a directory for changes.

Format Calling Sequence:

EXTRN DosFindNotifyFirst:FAR

```
PUSH@ ASCIIZ PathSpec ; Path spec
PUSH@ WORD DirHandle ; Path handle
PUSH WORD Attribute ; Search attribute
PUSH@ OTHER ResultBuf ; Result buffer
PUSH WORD ResultBufLen ; Result buffer length
PUSH@ WORD ChangeCount ; # of changes required
PUSH WORD FileInfoLevel ; File data required
PUSH DWORD Timeout ; Duration of call
PUSH DWORD 0 ; Reserved (must be zero)
CALL DosFindNotifyFirst
```

Where PathSpec is the path name to be monitored for directory changes.

DirHandle is the handle returned by OS/2 which is associated with this request.

FileInfoLevel is the level of file information required. A value of 0x0001 returns no information to ResultBuf. Values other than 0x0001 are not supported in this release of OS/2.

Attribute is the file attribute used in qualifying the files and/or directories to be monitored. Attribute cannot specify the volume label. Volume labels may be queried using DosQFsInfo.

ResultBuf is where the filesystem returns the results of the PathSpec monitoring.

ResultBufLen is the length of ResultBuf.

ChangeCount is the number of changes required to directories or files that match the PathSpec target and Attribute. The file system uses this field to return the number of changes that actually occurred.

Timeout is the maximum duration of DosFindNotifyFirst before returning to the function caller.

Returns: IF ERROR (AX not = 0)

AX = Error Code:

\* ERROR\_NO\_MORE\_SEARCH\_HANDLES - too many search handles are

currently in use by the process.

- \* ERROR\_PATH\_NOT\_FOUND - the drive letter is invalid, there are too many .., there are wildcards present before the last component, or a component of the directory path is not present.
- \* ERROR\_INVALID\_PARAMETER - the search attribute is invalid or a change count of zero is specified.
- \* ERROR\_BUFFER\_OVERFLOW - the specified buffer is too small to support a single entry.
- \* ERROR\_FINDNOTIFY\_TIMEOUT - the timeout expired without the requested number of changes occurring.
- \* ERROR\_VOLUME\_CHANGED - the volume changed in the drive specified PathSpec.
- \* Device-driver/device-manager errors listed "-----" on page ---.

Remarks The find notify API allow a desktop to be informed of changes in a directory in order to provide an up-to-date view of the directory. Changes consist of changes to names within a directory (file create, file delete, file/directory rename, directory create, directory removal) plus operations that change a file attribute. Those operations are SetFileMode, SetFileInfo, and SetPathInfo for both standard attributes and extended attributes. Changes that occur to PathSpec in between calls to DosFindNotifyFirst and DosFindNotifyNext or DosFindNotifyNext and DosFindNotifyNext will be included in the ChangeCount.

Changes to date and time and size will NOT be reported for each read or write, but will instead be reported when a file is closed or committed.

Volume changes will on the drive specified in PathSpec will be detected. Any thread waiting on a path on that drive will be returned with an error code. The ChangeCount will be valid and the corresponding information will be stored in ResultBuf. No DirHandle will be allocated and no call to DosFindNotifyClose should be issued.

DosFindNotifyFirst will block until either the timeout specified has elapsed or until the specified number of changes has been found.

00080

# 2.1.8.6.20 DosFindNotifyNext - Return directory: change information.

Purpose Return directory change information.

Format Calling Sequence:

EXTRN DosFindNotifyNext:FAR

```
PUSH WORD DirHandle ; Directory handle
PUSHM OTHER ResultBuf ; Result buffer
PUSH WORD ResultBufLen ; Result buffer length
PUSHM WORD ChangeCount ; # of changes required
PUSH DWORD Timeout ; Duration of call
CALL DosFindNotifyNext
```

Where DirHandle is the handle (previously returned by DOS) associated with a previous DosFindNotifyFirst or DosFindNotifyNext function call.

ResultBuf is where the FSD returns the results of the qualified directory search. The information returned is guaranteed to be at least as accurate as the most recent DosClose or DosBufReset.

ResultBufLen is the length of ResultBuf.

ChangeCount is the number of changes required to directories or files that match the PathSpec target and Attribute specified during a related, previous DosFindNotifyFirst. The file system uses this field to return the number of changes that actually occurred since the present DosFindNotifyNext was issued.

Timeout is the maximum duration of DosFindNotifyFirst before returning to the function caller.

Returns: IF ERROR (AX not = 0)

AX = Error Code:

- \* ERROR\_INVALID\_HANDLE - the specified handle is not in use.
- \* ERROR\_INVALID\_PARAMETER - a change count of zero is specified.
- \* ERROR\_BUFFER\_OVERFLOW - the specified buffer is too small to support a single entry.
- \* ERROR\_FINDNOTIFY\_TIMEOUT - the timeout expired without the requested number of changes occurring.
- \* ERROR\_VOLUME\_CHANGED - the volume changed in the drive specified in PathSpec.

2.0 Functional Characteristics

151

- \* ERROR\_FINDNOTIFY\_HANDLE\_IN\_USE - another thread is using the specified DirHandle.
- \* ERROR\_FINDNOTIFY\_HANDLE\_CLOSED - a DosFindNotifyClose has been issued on the DirHandle.
- \* Device-driver/device-manager errors listed "-----" on page ---.

Remarks DosFindNotifyNext will block until either the timeout specified has elapsed or until the specified number of changes has been found.

Volume changes will on the drive specified in PathSpec will be detected. Any thread waiting on a path on that drive will be returned with an error code. The ChangeCount will be valid and the corresponding information will be stored in ResultBuf. If no thread is waiting, the DirHandle associated with the drive will be marked invalid and the next call using the DirHandle will be returned immediately with an error code. A DosFindNotifyClose should be issued by the user.

Only one thread will be able to sleep on a DirHandle at one time. For example, a DosFindNotifyNext cannot be issued by one thread while another thread has issued a DosFindNotifyNext on the same DirHandle.

2.0 Functional Characteristics

152

00081

2.1.8.6.21 DosFsAttach - Creates and destroys FSD connections.:

\* Purpose Attaches or detaches drive to/from a remote FSD, or a pseudo-character device name to/from a local or remote FSD

Format Calling Sequence:

EXTRN DosFsAttach:FAR

```
PUSH@ ASCIIZ DeviceName ; Device name or 'd:'
PUSH@ ASCIIZ FSDName ; FSD name
PUSH@ OTHER DataBuffer ; Attach argument data
PUSH WORD DataBufferLen ; Buffer length
PUSH WORD OpFlag ; Attach or detach
PUSH DWORD 0 ; Reserved (must be zero)
CALL DosFsAttach
```

Where DeviceName points to either the drive letter followed by a colon, or points to a pseudo-character device name. If DeviceName is a drive, it is an ASCIIZ string having the form of drive letter followed by a colon. If an attach is successful, all requests to that drive are routed to the specified FSD. If a detach is successful, the drive will disappear from the system's name space.

If DeviceName is a pseudo-character device name (i.e., single file device), its format is that of an ASCIIZ string in the format of an OS/2 filename in a subdirectory called \DEV\ . All requests to that name are routed to the specified FSD after a successful attach. A successful detach removes the name from the system's name space.

FSDName is the ASCIIZ name of the remote FSD to attach to or detach from DeviceName.

DataBuffer points to the user-supplied FSD argument data area. The meaning of the data is specific to the FSD. The Databuffer will contain contiguous ascii strings, with the first word of the buffer containing the number of ascii strings.

DataBufferLen is the byte length of the data buffer.

OpFlag defines the type of operation to be performed.

- \* Attach = 0
- \* Detach = 1

Returns: AX = 0

ReturnCode = 0 if no error

ReturnCode = Error code if error

- \* ERROR\_INVALID\_DRIVE - the drive specified in DeviceName is illegal.
- \* ERROR\_INVALID\_PATH - the pseudodevice specified in DeviceName is illegal.
- \* ERROR\_INVALID\_FSD\_NAME - the FSD name specified is not found.
- \* ERROR\_INVALID\_LEVEL - the value of OpFlag is invalid.
- \* ERROR\_NOT\_ENOUGH\_MEMORY - cannot create OS/2 internal data structures.

Remarks The redirection of drive letters representing local drives is not supported.

FSDs that wish to establish open connections that are not attached to a name in the system's name space, for such purposes as optimizing UNC connections or establishing access rights, must use an DosFsCtl function to do so. DosFsAttach only creates attachments to drives or devices in the system's name space.

See description of pseudo-character devices.

#### 2.1.8.6.22 DosFsCtl - File System Control:

**Purpose** Allows an extended standard interface between an application and an FSD.

**Format** Calling Sequence:

EXTRN DosFsCtl:FAR

```
PUSH@ OTHER DataArea ; Data area
PUSH WORD DataLengthMax ; Max length of data area to be returned
PUSH @WORD DataLength ; In: Length of data returned by FSD
 ; Out: Length of data returned by FSD
PUSH@ OTHER ParmList ; Parameter list
PUSH WORD ParmLengthMax ; Max length of Parameter list passed to FSD
PUSH @WORD ParmLength ; In: Length of Parameter list sent by application
 ; Out: Length of Parameter returned by FSD
PUSH WORD FunctionCode ; Function code
PUSH@ ASCII2 RouteName ; Path or FSD name
PUSH WORD FileHandle ; File handle
PUSH WORD RouteMethod ; Method for routing.
PUSH DWORD 0 ; Reserved (must be zero)
CALL DosFsCtl
```

+ Where DataArea is a data area.

+ DataLengthMax is the maximum length of the data to be returned by the FSD in DataArea. DataLength may be longer than this on input, but not on output.

+ DataLength is the length in bytes of data returned in DataArea. On input, it is the length of data being sent, and on output is the length of the data that was returned by the FSD. If ERROR\_BUFFER\_OVERFLOW is returned from the call, then it is the size of the buffer required to hold the data returned by the FSD.

+ ParmList is a command specific parameter list.

+ ParmLengthMax is the maximum length in bytes of the data to be returned by the FSD in ParmList. ParmLength may be longer than this on input, but not on output.

+ ParmLength is, on input, the length of parameters passed in by the application, and on return, it is the length of parameters returned by the FSD. If ERROR\_BUFFER\_OVERFLOW is returned from the call, then it contains the size of buffer required to hold the parameters returned by the FSD. No other data is returned in this case.

FunctionCode is the filesystem-specific function code. Function

codes between 0x0000 and 0x7FFF are reserved for use by OS/2. For remote FSDs, there are two kinds of possible FsCtl calls:

ones that are handled locally,

ones that are exported across the network.

If bit 0x4000 is set in the function code, then this indicates to the remote FSD that the function should be exported. The range of function codes are split up as follows:

0x0000 - 0x7FFF reserved for definition by OS/2

0x8000 - 0x8FFF FSD defined fsctl functions handled by local FSD.

0xC000 - 0xFFFF FSD defined fsctl functions exported to server

\* FunctionCode = 0x0001 is used to obtain new error code information from the FSD.

\* FunctionCode = 0x0002 is used to query the FSD for the maximum size of individual EAs and the maximum size of the full EA list that it supports. The information is returned in DataArea in the following format:

```
EASizeBufStruc {
 unsigned short easb_MaxEASize /* Max. size of an ind
 EA supported */
 unsigned long easb_MaxEAListSize /* Max. full EA list
 supported */
}
```

RouteName is the ASCII2 name of the FSD, or the pathname of a file or directory the operation should apply to.

FileHandle is the file or device specific handle.

RouteMethod selects how the request will be routed.

\* RouteMethod = 1: FileHandle directs routing. RouteName must be NUL pointer (0L). The FSD associated with the handle will receive the request.

\* RouteMethod = 2: RouteMethod refers to a pathname, which directs routing. FileHandle must be -1. The FSD associated with the drive the pathname refers to at the time of the request will receive the request. The pathname need not refer to a file or directory which actually exists, only to a drive which does. A relative pathname may be used, it will be processed like any other pathname.



- \* RouteMethod = 3: RouteMethod refers to an FSD name, which directs routing. FileHandle must be -1. The named FSD will receive the request.

Returns: IF ERROR (AX not = 0)

AX = Error Code:

- \* ERROR\_INVALID\_FUNCTION - the specified function is illegal for the particular category and handle.
- \* ERROR\_INVALID\_CATEGORY - the specified category is illegal for the particular function and handle.
- \* ERROR\_INVALID\_PARAMETER - the filehandle != -1 when the routemethod == 2 or the routename != 0 when the routemethod == 1.
- \* ERROR\_INVALID\_HANDLE - the specified file handle is not in use or is attached to a physical device.
- \* ERROR\_INVALID\_FSD\_NAME - the specified FSDName is not responsible for managing the specified file handle.
- \* ERROR\_INVALID\_LEVEL - invalid route method.
- \* ERROR\_INTERRUPT - the current thread received a signal.
- \* ERROR\_BUFFER\_OVERFLOW - either DataArea or ParmList are not big enough to hold the data that the FSD is returning.

Remarks none

#### 2.1.8.6.23 DosMkDir - Make Subdirectory:

Purpose Creates the specified directory.

Format Calling Sequence:

EXTRN DosMkDir:FAR

PUSH@ ASCIIZ DirName ; New directory name  
PUSH DWORD 0 ; Reserved (must be zero)  
CALL DosMkDir

EXTRN DosMkDir2:FAR

PUSH@ ASCIIZ DirName ; New directory name  
PUSH@ OTHER EABuf ; Extended attribute buffer  
PUSH DWORD 0 ; Reserved (must be zero)  
CALL DosMkDir2

Where DirName is the ASCIIZ directory path name.

EABuf contains, on input, an EAOP structure. fpGEAList is ignored. fpFEAList points to a data area where the relevant FEA list is to be found. offError is ignored.

On output, fpGEAList is unchanged. fpFEAList is unchanged as is the area pointed to by fpFEAList. If an error occurred during the set, offError will be the offset of the FEA where the error occurred. The API return code will be the error code corresponding to the condition generating the error. If no error occurred, offError is undefined.

If EABuf is 0x00000000, then no extended attributes are defined for the directory.

Returns: IF ERROR (AX not = 0)

AX = Error Code:

- \* ERROR\_PATH\_NOT\_FOUND - the drive letter is invalid, there are too many ..., there are wildcards present, or a component of the directory path is not present.
- \* ERROR\_FILENAME\_EXCED\_RANGE - the path specified is unacceptable to the FSD managing the volume.
- \* ERROR\_ACCESS\_DENIED - the directory already exists or a file with this name exists.

- \* Device-driver/device-manager errors listed "-----  
-----" on page ---.
- \* ERROR\_INVALID\_EA\_NAME - there is an illegal character in an EA name. The offError field points to the offending FEA.
- \* ERROR\_EA\_LIST\_INCONSISTENT - the cbList does not match the sum of the lengths of the FEA structures.
- \* ERROR\_EA\_VALUE\_UN SUPPORTABLE - the FSD detects an error in the value portion of an EA. The offError field points to the offending FEA.

Remarks If any member of the directory path does not exist, then the directory path is not created. On return, a new directory is created at the end of the specified path.

DosSysInfo must be used by an application to determine the maximum path length supported by OS/2. The returned value should be used to dynamically allocate buffers that are to be used to store paths. This will ensure that applications function correctly (wrt path lengths) for future versions of OS/2. The path length includes the drive specifier (i.e. d:), the leading '\', and the trailing NUL.

#### 2.1.8.6.24 DosMove - Move a File or a Subdirectory:

Purpose Moves the specified file or subdirectory.

Format Calling Sequence:

EXTRN DosMove:FAR

PUSH@ ASCIIZ OldPathName ; Old path name  
PUSH@ ASCIIZ NewPathName ; New path name  
PUSH DWORD 0 ; Reserved (must be zero)  
CALL DosMove

Where

OldPathName is the old ASCIIZ path name of the file or subdirectory to be moved.

NewPathName is the new ASCIIZ path name of the file or subdirectory to be moved.

Returns: IF ERROR (AX not = 0)

AX = Error Code:

- \* ERROR\_PATH\_NOT\_FOUND - the drive letter is invalid, there are too many ..., there are wildcards present, or a component of the directory path is not present.
- \* ERROR\_FILENAME\_EXCED\_RANGE - the path specified is unacceptable to the FSD managing the volume.
- \* ERROR\_SHARING\_VIOLATION - the OldPathName is currently open.
- \* ERROR\_FILE\_NOT\_FOUND - the OldPathName is not present
- \* ERROR\_NOT\_SAME\_DEVICE - the OldPathName and NewPathName are on different physical devices.
- \* ERROR\_ACCESS\_DENIED - NewPathName already exists, OldPathName is a character device, or OldPathName is a pseudocharacter device.
- \* ERROR\_CIRCULARITY\_REQUESTED - the OldPathName is an ancestor of the NewPathName.
- \* ERROR\_DIRECTORY\_IN\_CDS - the OldPathName is the current directory of a process.

- \* ERROR\_SHARING\_BUFFER\_EXCEEDED - there is not enough memory to hold sharing information.
- \* Device-driver/device-manager errors listed "-----" on page ---.

Remarks The directory paths need not be the same, allowing a file or subdirectory to be moved to another directory and renamed in the process.

\* Wildcard characters are not allowed in the source or destination name.

File systems will reject requests that move a parent directory to one of its subdirectories in order to avoid a loop in the directory tree. A subdirectory cannot be both a descendent and an ancestor of the same directory.

An attempt to move the current directory or any of its ancestors for the current or any other process will fail and cause the operation to terminate.

An attempt to move the current directory for any process will fail and cause the operation to terminate.

Read-only files in the target path are not replaced.

DosQSysInfo must be used by an application to determine the maximum path length supported by OS/2. The returned value should be used to dynamically allocate buffers that are to be used to store paths. This will ensure that applications function correctly (wrt path lengths) for future versions of OS/2. The path length includes the drive specifier (i.e. d:), the leading '\', and the trailing NUL.

DosMove will move the source's attributes (date of creation, time of creation, ...) to the target.

#### 2.1.8.6.25 DosNewSize - Change File's Size:

Purpose Changes a file's logical (EOD) size.

Format Calling Sequence:

EXTRN DosNewSize:FAR

PUSH WORD FileHandle ; File handle  
PUSH DWORD FileSize ; File's new size  
CALL DosNewSize

Where

FileHandle is the handle of the file whose size is being changed.

FileSize is the file's new size in bytes.

Returns: IF ERROR (AX not = 0)

AX = Error Code:

- \* ERROR\_DISK\_FULL - there is not enough room to grow the file.
- \* ERROR\_INVALID\_PARAMETER - the size is negative.
- \* ERROR\_INVALID\_HANDLE - the handle is not in use or is attached to a physical device.
- \* ERROR\_ACCESS\_DENIED - this handle is opened read-only.
- \* ERROR\_LOCK\_VIOLATION - the region of file growth overlaps a locked region.
- \* Device-driver/device-manager errors listed "-----" on page ---.

Remarks The file handle must be opened for read/write or write-only access.

The value of new bytes in the extended file is undefined.

The file system will make a reasonable attempt to allocate the new size in a contiguous (or nearly contiguous) space on the media.

#### 2.1.8.6.26 DosOpen - Open a File:

**Purpose** Creates the specified file (if necessary), and opens it.

**Format** Calling Sequence:

EXTRN DosOpen:FAR

```
PUSH# ASCII2 FileName ; File path name
PUSH# WORD FileHandle ; New file's handle
PUSH# WORD ActionTaken ; Action taken
PUSH# DWORD FileSize ; File primary allocation
PUSH# WORD FileAttribute ; File Attribute
PUSH# WORD OpenFlag ; Open function type
PUSH# WORD OpenMode ; Open mode of the file
PUSH# DWORD 0 ; Reserved (must be zero)
CALL DosOpen
```

EXTRN DosOpen2:FAR

```
PUSH# ASCII2 FileName ; File path name
PUSH# WORD FileHandle ; New file's handle
PUSH# WORD ActionTaken ; Action taken
PUSH# DWORD FileSize ; File primary allocation
PUSH# WORD FileAttribute ; File Attribute
PUSH# WORD OpenFlag ; Open function type
PUSH# DWORD OpenMode ; Open mode of the file
PUSH# OTHER EABuf ; EA buffer
PUSH# DWORD 0 ; Reserved (must be zero)
CALL DosOpen2
```

**Where**

FileName is the ASCII2 path name of the file or device name to be opened.

FileHandle is where the system returns the file handle.

ActionTaken is where the system returns a description of the action taken as a result of the DosOpen function call.

- \* 0x0001 file existed
- \* 0x0002 file was created
- \* 0x0003 file was replaced

FileSize is the new file's logical size (EOD) in bytes.

FileAttribute is the file attribute. Refer to DosQFileMode or DosSetFileMode for a description of FileAttribute.

OpenFlag specifies the action to be taken depending on whether or not the file exists.

- \* OpenFlag specification:

Low Order Byte

|             |                             |
|-------------|-----------------------------|
| * ---- xxxx | action taken if file exists |
| * ---- 0000 | fail                        |
| * ---- 0001 | open file                   |
| * ---- 0010 | replace file                |
| xxxx ----   | action taken if file        |
|             | doesn't exist               |
| 0000 ----   | fail                        |
| 0001 ----   | create file                 |

OpenMode is the open mode and consists of the following bit fields. They are the:

- \* Inheritance flag
- \* Write-through flag
- \* Fail-errors flag
- \* Sharing mode field
- \* Access field
- \* Reserved bit fields

The bit field mapping is shown as follows:

|           |                                 |
|-----------|---------------------------------|
|           | 1 1 1 1 1 1                     |
| Open Mode | 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 |
| bits      | D W F C R L L L I S S S R A A A |

D Direct Open

The file is opened as follows:

If D = 0; FileName represents a file to be opened in the normal way.  
If D = 1; FileName is "<Drive>:" and represents a mounted disk or diskette volume to be opened for direct access.

## W File Write-through

The file is opened as follows:

If W = 0; Writes to the file may be run through the DOS buffer cache.  
If W = 1; Writes to the file may go through the DOS buffer cache but sectors will be written (actual file I/O completed) before a synchronous write call returns. This state of the file defines it as a synchronous file.

NOTE that this is a mandatory bit in that the data MUST be written out to the medium for synchronous write operations. This bit is not inherited by child processes.

## I Inheritance Flag

If I = 0; File handle is inherited by a spawned process resulting from a DosExecPgm call.  
If I = 1; File handle is private to the current process.

This bit is not inherited by child processes.

## F Fail-Errors

Media I/O errors will be handled as follows:

If F = 0; reported through the system critical error handler.  
If F = 1; reported directly to the caller via return code.

This bit is not inherited by child processes. Media I/O errors generated through an IoCtl Category 8 function, always get reported directly to the caller via return code. The Fail-Errors functionality applies only to non-IoCtl handle-based type file I/O calls.

## C Cache/No-Cache

The file is opened as follows:

If C = 0; I/O to the file need not be done through the disk driver cache.  
If C = 1; It is advisable for the disk driver

to cache the data in I/O operations on this file.

This bit is an ADVISORY bit, and is used to advise FSDs and device drivers on whether it is worth caching the data or not. This bit, like the write-through bit, is a per-handle bit. It is not inherited by child processes.

## R Reserved and must-be-zero field

## L Locality of reference

The locality-of-reference is advisory information about how the application will access the file.

If L == 00; no locality known  
If L == 01; mainly sequential access  
If L == 10; mainly random access  
If L == 11; random with some locality

## S Sharing Mode

The file sharing mode field defines what operations other processes may perform on the file.

If S = 001; Deny Read/Write access  
If S = 010; Deny Write access  
If S = 011; Deny Read access  
If S = 100; Deny Neither Read or Write access (Deny None)

Any other value is invalid.

## A Access Mode

The file access is assigned as follows:

If A = 000; Read-only access  
If A = 001; Write-only access  
If A = 010; Read/Write access

Any other value is invalid.

Any other combinations are invalid.

When opening a file, it is important to inform OS/2 what operations other processes may perform on this file (sharing mode). If it is permissible for other processes to continue to read this file while your process is operating on the file, you should specify Deny Write, which inhibits writing by other

processes, but allows reading by them.

Similarly, it is important to specify what operations your process will perform (access mode). The Read/Write access mode causes the open request to fail if another process has the file opened with any sharing mode other than deny none. If however, all you intended to do is read from the file, your open will not succeed unless all other processes have specified deny none or deny write (therefore increasing access to the file). File sharing requires cooperation of both sharing processes. This cooperation is communicated through the sharing and access mode.

EABuf contains, on input, an EAOP structure. fpGEAList is ignored. fpFEAList points to a data area where the relevant FEA list is to be found. offError is ignored.

On output, fpGEAList is unchanged. fpFEAList is unchanged as is the area pointed to by fpFEAList. If an error occurred during the set, offError will be the offset of the FEA where the error occurred. The API return code will be the error code corresponding to the condition generating the error. If no error occurred, offError is undefined.

If EABuf is 0x00000000, then no extended attributes are defined for the file.

If Extended Attributes are not to be defined or modified, then the pointer, EABuf, must be set to null.

Returns: IF ERROR (AX not = 0)

AX = Error Code:

- \* ERROR\_INVALID\_PARAMETER - the OpenFlag value is invalid, the OpenMode value is invalid, the OpenFlag/OpenMode values are incompatible, or the size specified in a replace/create operation is negative.
- \* ERROR\_INVALID\_ACCESS - the access mode field is invalid.
- \* ERROR\_ACCESS\_DENIED - the attributes specified are invalid, the file is read only and OpenMode requests a write access, or the FileName points to a directory.
- \* ERROR\_OPEN\_FAILED - the combination of OpenFlag and state of the file results in a failed operation.
- \* ERROR\_DISK\_FULL - there is not enough room on the media to create the file with the specified size.
- \* ERROR\_TOO\_MANY\_OPEN\_FILES - there are too many files open by either the system or the process.

- \* ERROR\_FILE\_NOT\_FOUND - the last component of the file name is not present on the media.
- \* ERROR\_PATH\_NOT\_FOUND - a component of the directory path is not present.
- \* ERROR\_SHARING\_BUFFER\_EXCEEDED - there is not enough memory to hold sharing information.
- \* ERROR\_SHARING\_VIOLATION - the FileName is currently open.
- \* ERROR\_FILENAME\_EXCED\_RANGE - the path specified is unacceptable to the FSD managing the volume.
- \* Device-driver/device-manager errors listed "-----" on page ---.
- \* ERROR\_INVALID\_EA\_NAME - there is an illegal character in an EA name. The offError field points to the offending FEA.
- \* ERROR\_EA\_LIST\_INCONSISTENT - the cbList does not match the sum of the lengths of the FEA structures.
- \* ERROR\_EA\_VALUE\_UNSUPPORTED - the FSD detects an error in the value portion of an EA. The offError field points to the offending FEA.
- \* ERROR\_DEVICE\_IN\_USE - the device is in use by another application.

Remarks The read/write pointer is set at the first byte of the file. The read/write pointer can be changed through function DosChgFilePtr.

The file's date and time can be obtained through function DosQFileInfo.

The file's date and time can be set through function DosSetFileInfo, and its attribute can be obtained through function DosQFileMode.

The FileSize parameter affects the size of the file only when it is created or replaced. If an existing file is simply opened, FileSize is ignored. This is a recommended size for the file. If allocation of the full size fails, the open may still succeed.

The Direct Open bit parameter is the "Direct I/O flag". It provides an access mechanism to an entire disk or diskette volume independent of the file system. This mode of opening the volume currently mounted on the drive, is used to return a handle to the caller which represents the logical volume as a single file. In

order to block other processes from accessing the logical volume, the caller must also issue a DosDevioCtl Category 8 sub-function 0, which requires the file handle for the logical volume returned by DosOpen.

The file handle state bits can be set by the DosOpen and DosSetFHState function calls. An application may query the file handle state bits as well as the rest of the Open Mode field, by using the DosQFHandState function call.

The returned file handle must be used for subsequent input and output to the file.

The value of new bytes in the extended file is undefined.

The file system must make a reasonable attempt to allocate the new size in a contiguous (or nearly contiguous) area on the media. Extended attributes that indicate required contiguity may reject the call if unable to allocate contiguous space.

The extended attributes will be set for creation of a new file, replacement of an existing file, and truncation of an existing file. No attributes are set for a normal open.

FileAttribute cannot be set to Volume Label. Volume labels cannot be opened.

Notes:

1. A multitasking system must be able to use files and their existence as semaphores. This function call may be used as a test-and-set semaphore when used to create a new file.
2. When a file is closed, any sharing restrictions placed on it by the open are removed.
3. The file read-only attribute can be set by using the DosSetFileMode function call or the OS/2 ATTRIB command.
4. If the file is inherited by a spawned process, all sharing and access restrictions are also inherited.
5. If an open file handle is duplicated by function call DosDupHandle, all sharing and access restrictions are also duplicated.

Sharing Modes

Deny Read/Write Mode

If a file is successfully opened in Deny Read/Write mode, access to the file is exclusive. A file currently open in this mode cannot be opened again in any sharing mode by any process (including the current process) until the file is closed.

Deny Write Mode

A file successfully opened in Deny Write sharing mode, prevents any other write access opens to the file (A = 001 or 010) until the file is closed. An attempt to open a file in Deny Write mode is unsuccessful if the file is currently open with a write access.

Deny Read Mode

A file successfully opened in Deny Read sharing mode, prevents any other read sharing access opens to the file (A = 000 or 010) until the file is closed. An attempt to open a file in Deny Read sharing mode is unsuccessful if the file is currently open with a read access.

Deny None Mode

A file successfully opened in Deny None mode, places no restrictions on the read/write accessibility of the file.

Named Pipe Considerations Opens the client end of a pipe by name and returns a handle. The pipe must be in listen state for the open to succeed; otherwise the open fails with PIPE BUSY (this happens, for example, if all instances of the pipe are already open, if the pipe is closed but not yet disconnected by the serving end, or if no DosConnectNmPipe has been issued against the pipe since it was last disconnected). Once a given instance has been opened by a client that same instance cannot be opened by another client at the same time (i.e., pipes can only be two-ended at present); the opening process can of course dup the open handle as many times as desired. The access and sharing modes specified on the open must be consistent with those specified on the DosMakeNmPipe. Pipes are always opened with the pipe-specific states set to B = 0 (blocking reads/writes), RR = 00 (read as byte stream). The client can change these modes via DosSetPHandState if desired.

2.1.8.6.27 DosQCurDir - Query Current Directory:

**Purpose** Gets the full path name of the current directory for the requesting process for the specified drive.

**Format** Calling Sequence:

EXTRN DosQCurDir:FAR

PUSH WORD DriveNumber ; Drive number

PUSH@ OTHER DirPath ; Directory path buffer

PUSH@ WORD DirPathLen ; Directory path buffer length

CALL DosQCurDir

**Where**

DriveNumber is the drive number (0=default, 1=A, ... ).

DirPath is where the system returns the full directory path name.

DirPathLen is the length of the DirPath buffer.

**Returns:** IF ERROR (AX not = 0)

AX = Error Code:

- \* ERROR\_INVALID\_DRIVE - the specified drive is invalid.
- \* ERROR\_BUFFER\_OVERFLOW - the current directory is too long to fit into the specified directory.
- \* Device-driver/device-manager errors listed "-----" on page ----.

**Remarks** The drive letter is not part of the returned string. The string does not begin with a backslash and is terminated by a byte containing 0x00.

The system returns the length of the returned DirPath string (not including the 0x00 terminating null byte) in DirPathLen.

In the case where the DirPath buffer is of insufficient length to hold the current directory path string, the system returns the required length (in bytes) for DirPath in DirPathLen.

DosQSysInfo must be used by an application to determine the maximum path length supported by OS/2. The returned value should be used to dynamically allocate buffers that are to be used to store paths. This will ensure that applications function correctly (wrt path lengths) for future versions of OS/2. The path length includes the drive specifier (i.e. d:), the leading

'\' and the trailing NUL.



#### 2.1.8.6.28 DosQCurDisk - Query Current Disk:

Purpose Determine the current default drive for the requesting process.

Format Calling Sequence:

EXTRN DosQCurDisk:FAR

PUSH0 WORD DriveNumber ; Default drive number  
PUSH0 DWORD LogicalDriveMap ; Drive-map area  
CALL DosQCurDisk

Where

DriveNumber is where the system returns the number of the default drive (1-A, 2-B, ...)

LogicalDriveMap is a bit map (stored in the low-order portion of the 32-bit double word area) in which the system returns the mapping of the logical drives. Logical Drives A-Z have a one-to-one mapping with the bit positions 0-25 of the map.

- \* If bit value = 0, the logical drive does not exist
- \* If bit value = 1, the logical drive exists

Returns: No error returns are defined for this API.

#### 2.1.8.6.29 DosQFHandState - Query File Handle State:

Purpose Query the state of the specified file.

Format Calling Sequence:

EXTRN DosQFHandState:FAR

PUSH WORD FileHandle ; File handle  
PUSH0 WORD FileHandleState ; File handle state  
CALL DosQFHandState

Where

FileHandle is the handle of the file to be queried.

FileHandleState is the file handle state and consists of the following bit fields. They are the:

- \* Inheritance flag
- \* Write-through flag
- \* Fail-errors flag
- \* Sharing mode field
- \* Access field
- \* Reserved bit fields

The bit field mapping is shown as follows:

Open Mode bits 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0  
D W F C R R R R I S S S R A A A

D Direct Open

The file is opened as follows:

- If D = 0; Pathname represents a file opened in the normal way.
- If D = 1; Pathname is "<Drive>:" and represents a mounted disk or diskette volume to opened for direct access.

I Inheritance Flag

- If I = 0; File handle is inherited by a spawned process resulting

Microsoft Confidential  
OS/2 1.2 IFS Patent Documentation

from a DosExecPgm call.  
If I = 1; File handle is private to the  
current process.

This bit is not inherited by child processes.

W File Write-through

The file is opened as follows:

If W = 0; Writes to the file may be run  
through the DOS buffer cache.

If W = 1; Writes to the file may go  
through the DOS buffer  
cache but sectors  
will be written (actual file  
I/O completed) before a  
synchronous write call returns.  
This state of the file defines  
it as a synchronous file.

NOTE that this is a mandatory bit in that the data MUST be  
written out to the medium for synchronous write  
operations. This bit is not inherited by child processes.

F Fail-Errors

Media I/O errors will be handled as follows:

If F = 0; reported through the system  
critical error handler.

If F = 1; reported directly to the  
caller via return code.

This bit is not inherited by child processes. Media I/O  
errors generated through an Ioctl Category 8 function,  
always get reported directly to the caller via return  
code. The Fail-Errors functionality applies only to  
non-ioctl handle-based type file I/O calls.

C Cache/No-Cache

The file is opened as follows:

If C = 0; I/O to the file need not be done  
through the disk driver cache.

If C = 1; It is advisable for the disk driver  
to cache the data in I/O operations  
on this file.

This bit is an ADVISORY bit, and is used to advise FSDs  
and device drivers on whether it is worth caching the data

Microsoft Confidential  
OS/2 1.2 IFS Patent Documentation

or not. This bit, like the write-through bit, is a  
per-handle bit. It is not inherited by child processes.

R These bits are reserved and should be set to the values  
returned by DosQFHandState in these positions.

S Sharing Mode

The file sharing mode field defines what operations  
other processes may perform on the file.

If S = 001; Deny Read/Write access

If S = 010; Deny Write access

If S = 011; Deny Read access

If S = 100; Deny Neither Read or  
Write access (Deny None)

Any other value is invalid.

A Access Mode

The file access is assigned as follows:

If A = 000; Read-only access

If A = 001; Write-only access

If A = 010; Read/Write access

Any other value is invalid.

Returns: IF ERROR (AX not = 0)

AX = Error Code:

\* ERROR\_INVALID\_HANDLE - the handle is not in use or is  
attached to a physical device.

Remarks

Named Pipe Considerations As defined by OS/2. D = 0. Other bits as  
defined by DosMakeNmPipe (serving end), DosOpen (client end), or  
the last DosSetFHandState.

#### 2.1.8.6.30 DosQFileInfo - Query a File's Information:

**Purpose** Returns information for a specific file.

**Format** Calling Sequence:

EXTRN DosQFileInfo:FAR

```
PUSH WORD FileHandle ; File handle
PUSH WORD FileInfoLevel ; File data required
PUSH# OTHER FileInfoBuf ; File data buffer
PUSH WORD FileInfoBufSize ; File data buffer size
CALL DosQFileInfo
```

**Where** FileHandle is the file handle.

FileInfoLevel is the level of file information required.

File information, where applicable, is at least as accurate as the most recent DosClose, DosBufReset, or DosSetFileInfo.

Level 0x0001 information is returned in the following format:

```
struct {
 unsigned short dateCreate;
 unsigned short timeCreate;
 unsigned short dateAccess;
 unsigned short timeAccess;
 unsigned short dateWrite;
 unsigned short timeWrite;
 long cbEOF;
 long cbAlloc;
 unsigned short attr;
};
```

Level 0x0002 simply adds cbList to level 0x0001, returning a structure in the following format:

```
struct {
 unsigned short dateCreate;
 unsigned short timeCreate;
 unsigned short dateAccess;
 unsigned short timeAccess;
 unsigned short dateWrite;
 unsigned short timeWrite;
 long cbEOF;
 long cbAlloc;
 unsigned short attr;
 unsigned long cbList;
};
```

Level 0x0003 file information returns a subset of the EA information for the file. On input, FileInfoBuf is an EAOP structure above. fpGEAList points to a GEA list defining the attribute names whose values will be returned. fpFEAList points to a data area where the relevant FEA list will be returned. The length field of this FEA list is valid, giving the size of the FEA list buffer. offError points to the offending GEA entry in case of error.

On output, FileInfoBuf is unchanged. The buffer pointed to by fpFEAList is filled in with the returned information.

If the buffer fpFEAList points to isn't large enough to hold the returned information (ie ERROR\_BUFFER\_OVERFLOW) cbList will still be valid, assuming there's at least enough space for it. Its value will be the size of the entire EA set for the file, even though only a subset of attributes was requested.

\*\*\*THE FOLLOWING UNPUBLISHED FUNCTION WILL GO AWAY IN FUTURE \*\*\*\*\*  
\*\*\*RELEASES AND SHOULD BE UNPUBLISHED\*\*\*

Level 0x0004 file information returns all EA info for the file. On input, FileInfoBuf is an EAOP structure above. fpGEAList contents are ignored. fpFEAList points to a data area where the relevant FEA list will be returned. The length field of this FEA list is valid, giving the size of the FEA list buffer. offError is ignored.

On output, FileInfoBuf is unchanged. The buffer pointed to by fpFEAList is filled with the returned information.

It is important that applications expecting to be compatible with future versions of OS/2 not use this infolevel, since the limit on the size of the full EA list will be raised. Thus, this level is documented for utility and system use only.

+ If the buffer fpFEAList points to isn't large enough to hold the  
+ returned information (ie ERROR\_BUFFER\_OVERFLOW) cbList will still  
+ be valid.

+ \*\*\*\*\*End of the UNPUBLISHED FUNCTION\*\*\*\*\*

FileInfoBuf is the storage area where the system returns the requested level of file information.

FileInfoBufSize is the length of FileInfoBuf.

Returns: IF ERROR (AX not = 0)

AX = Error Code:

- \* ERROR\_INVALID\_LEVEL - the specified FileInfoLevel is not valid.
- \* ERROR\_INVALID\_HANDLE - the handle is not in use or is attached to a physical device.
- \* ERROR\_DIRECT\_ACCESS\_HANDLE - the specified handle is opened with the direct open bit set.
- \* ERROR\_BUFFER\_OVERFLOW - the specified buffer is not long enough to return the desired information.
- \* Device-driver/device-manager errors listed "-----  
-----" on page ---.
- \* ERROR\_INVALID\_EA\_NAME - there is an illegal character in an EA name. The offError field points to the offending GEA.
- \* ERROR\_EA\_LIST\_INCONSISTENT - the cbList does not match the sum of the lengths of the GEA structures.
- \* ERROR\_ACCESS\_DENIED - the specified file is not open for read access.

+ Remarks The FAT file system supports only the modification date and time.  
+ Zero will be returned for the creation and access dates and times.

+ DosQFileInfo for all infolevels will work only for files opened in  
+ open-for-read or open-for-both/deny-write mode.

## 2.1.8.6.31 DosQFileMode - Query File Mode:

Purpose Query the mode (attribute) of the specified file.

Format Calling Sequence:

EXTRN DosQFileMode:FAR

PUSH# ASCIIZ FilePathName ; File path name  
PUSH# WORD CurrentAttribute ; Data area  
PUSH DWORD 0 ; Reserved (must be zero)  
CALL DosQFileMode

Where

FilePathName is the ASCIIZ file path name.

CurrentAttribute is where the system returns the file's current attribute.

Returns: IF ERROR (AX not = 0)

AX = Error Code:

- \* ERROR\_PATH\_NOT\_FOUND - a component of the directory path is not present.
- \* ERROR\_FILE\_NOT\_FOUND - the last component of the file name is not present on the media.
- \* ERROR\_FILENAME\_EXCED\_RANGE - the path specified is unacceptable to the FSD managing the volume.
- \* Device-driver/device-manager errors listed "-----  
-----" on page ---.

Remarks The 'Volume Label' type attribute is not returned by DosQFileMode, DosQFileInfo may be used for this purpose.

File attributes are defined as follows.

0x0001 read only file.  
0x0002 hidden file.  
0x0004 system file.  
0x0010 subdirectory.

0x0020 file 'archive.

All other attributes are reserved.

DosQSysInfo must be used by an application to determine the maximum path length supported by OS/2. The returned value should be used to dynamically allocate buffers that are to be used to store paths. This will ensure that applications function correctly (wrt path lengths) for future versions of OS/2. The path length includes the drive specifier (i.e. d:), the leading '\', and the trailing NUL.

#### 2.1.8.6.32 DosQFsAttach - Query attached FSD information:

**Purpose** Query information about an attached remote file system or a local file system or about a character device or about pseudo-character device name attached to a local or remote FSD.

**Format** Calling Sequence:

EXTRN DosQFsAttach:FAR

PUSH# ASCIIZ DeviceName ; Device name or 'd:'  
PUSH WORD Ordinal ; Ordinal of entry in name list  
PUSH WORD FSAInfoLevel ; Type of attached FSD data required  
PUSH# OTHER DataBuffer ; Returned data buffer  
PUSH# WORD DataBufferLen ; Buffer length  
PUSH DWORD 0 ; Reserved (must be zero)  
CALL DosQFsAttach

**Where** DeviceName points to the drive letter followed by a colon, or points to a character or pseudo-character device name, or is ignored for some values of FSAInfoLevel. If DeviceName is a drive, it is an ASCIIZ string having the form of drive letter followed by a colon. If DeviceName is a character or pseudo-character device name its format is that of an ASCIIZ string in the format of a OS/2 filename in a subdirectory called \DEV\.

Ordinal is an index into the list of character or pseudo-character devices, or the set of drives. Ordinal always starts at 1. The Ordinal position of an item in a list has no significance at all, Ordinal is used strictly to step through the list. The mapping from Ordinal to item is volatile, and may change from one call to DosQFsAttach to the next.

FSAInfoLevel is the level of information required, and determines which item the data in DataBuffer refers to.

Level 0x0001 returns data for the specific drive or device name referred to by DeviceName. The Ordinal field is ignored.

Level 0x0002 returns data for the entry in the list of character or pseudo-character devices selected by Ordinal. The DeviceName field is ignored.

Level 0x0003 returns data for the entry in the list of drives selected by Ordinal. The DeviceName field is ignored.

DataBuffer is the return information buffer, it is in the following format:

```
struct {
 unsigned short lType;
 unsigned short cbName;
 unsigned char szName[];
 unsigned short cbFSDName;
 unsigned char szFSDName[];
 unsigned short cbFSADData;
 unsigned char rgFSADData[];
};
```

lType      type of item

- \*    1 = Resident character device
- \*    2 = Pseudo-character device
- \*    3 = Local drive
- \*    4 = Remote drive attached to FSD

cbName      Length of item name, not counting null.

szName      Item name, ASCIIZ string.

cbFSDName   Length of FSD name, not counting null.

szFSDName   Name of FSD item is attached to, ASCIIZ string.

cbFSADData   Length of FSD Attach data returned by FSD.

rgFSADData   FSD Attach data returned by FSD.

\*    szFSDName is the FSD name exported by the FSD, which is not  
\*    necessarily the same as the FSD name in the boot sector.

\*    For local character devices (lType = 1), cbFSDName = 0 and  
\*    szFSDName will contain only a terminating NULL byte, and cbFSADData  
\*    = 0.

\*    For local drives (lType = 3), szFSDName will contain the name of  
\*    the FSD attached to the drive at the time of the call. This  
\*    information changes dynamically. If the drive is attached to the  
\*    kernel's resident file system, szFSDName will contain "FAT" or  
\*    "UNKNOWN". Since the resident file system gets attached to any  
\*    disk that other FSDs refuse to MOUNT, it is possible to have a  
\*    disk that does not contain a recognizable file system, but yet  
\*    gets attached to the resident file system. In this case, it is  
\*    possible to detect the difference, and this information would help  
\*    programs in not destroying data on a disk that was not properly  
\*    recognized.

DataBufferLen is the byte length of the return buffer. Upon return, it is the length of the data returned in DataBuffer by the FSD.

Returns:    IF ERROR (AX not = 0)

AX = Error Code:

- \*    ERROR\_INVALID\_DRIVE - the drive specified is invalid
- \*    ERROR\_BUFFER\_OVERFLOW - the specified buffer is too short for the returned data.
- \*    ERROR\_NO\_MORE\_ITEMS - the Ordinal specified refers to an item not in the list.
- \*    ERROR\_INVALID\_LEVEL - invalid info level

\* Remarks    Information about all block devices and all character and pseudo-character devices is returned by this call.

The information returned by this call is highly volatile. Calling programs should be aware the the returned information may have already changed by the time it's returned to them.

The information returned for disks that are attached to the kernel's resident file system can be used to determine if the kernel definitely recognized the disk as one with its file system on it, or if the kernel just attached its file system to it because no other FSDs MOUNTed the disk. This can be important information for a program that needs to know what file system is attached to the drive. It is quite easy to get into a situation where the FSD that recognizes a certain disk has not been installed into the system. In such a case, there is a potential for the data on the disk to be destroyed since the wrong file system will be attached to the disk by default.

#### 2.1.8.6.33 DosQFInfo - Query File System Information:

Purpose Query information from a file system.

Format Calling Sequence:

EXTRN DosQFInfo:FAR

```
PUSH WORD DriveNumber ; Drive number
PUSH WORD FSInfoLevel ; File system data level required
PUSH OTHER FSInfoBuf ; File system info buffer
PUSH WORD FSInfoBufSize ; File system info buffer size
CALL DosQFInfo
```

Where DriveNumber is the logical drive number (0=default, 1=A, 2=B, 3=C, ... ) and represents the FSD for the media currently in that drive or the FSD that is currently attached with that drive.

When a logical drive is specified, the media in the drive is examined (local drive only) and the request is passed to the FSD responsible for managing that media or to the FSD that is attached to the drive (remote case).

FSInfoLevel is the level of file information required.

Level 0x0001 information is returned in the following format:

```
struct {
 unsigned long reserved;
 unsigned long csecPerAlloc;
 unsigned long callocTotal;
 unsigned long callocFree;
 unsigned short cbPerSec;
};
```

Level 0x0002 information is returned in the following format:

```
struct {
 unsigned long ulVSN;
 unsigned char cbVollLabel;
 unsigned char szVollLabel[];
};

ulVSN volume serial number
cbVollLabel length of volume label
```

szVollLabel asciiz volume label

FSInfoBuf is the storage area where the system returns the requested level of file system information.

FSInfoBufSize is the length of FSInfoBuf.

Returns: IF ERROR (AX not = 0)

AX = Error Code:

- \* ERROR\_INVALID\_DRIVE - the drive specified is invalid
- \* ERROR\_BUFFER\_OVERFLOW - the specified buffer is too short for the returned data.
- \* ERROR\_INVALID\_LEVEL - the specified FSInfoLevel is not supported.
- \* Device-driver/device-manager errors listed "-----" on page ---.

Remarks Trailing blanks supplied at volume label definition time are not considered to be part of the label and are therefore not returned as valid label data. Volume label is limited to a length of 11 bytes.

\* Volume Serial Number is a unique 32-bit number used by OS/2 to positively identify its disk/diskette volumes. The hard error daemon will prompt the user for an unmounted removable volume by displaying both the Volume Serial Number (as an 8 digit hexadecimal number) and the Volume Label.

\* If there is no volume serial number on the disk/diskette, the volume serial number information will be returned as binary zeros.

\* If there is no volume label on the disk/diskette, the volume label information will be returned as blank spaces.

\* If there is no volume serial number and/or volume label for disk/diskette volumes formatted by DOS 3.X, this information is not displayed by the Hard Error Handler.

#### 2.1.8.6.34 DosQHandType - Query a Handle Type:

**Purpose** Determines whether a handle references a file or a device.

**Format** Calling Sequence:

EXTRN DosQHandType:FAR

```
PUSH WORD FileHandle ; File handle
PUSH WORD HandType ; Handle type response
PUSH WORD FlagWord ; Device descriptor word
CALL DosQHandType
```

**Where**

FileHandle is the file handle.

HandType is where the system returns the value indicating the handle type. HandType is composed of two bytes:

HandleClass Describes the handle class. It may take on the following values in the low byte of HandType:

- \* 0, handle is for a disk file.
- \* 1, handle is for a character device.
- \* 2, handle is for a pipe.

Values greater than 2 are reserved.

HandleBits Provides further information about the handle in the high byte of HandType. This byte is broken into eight bits, whose meaning depends upon the value of HandleClass:

|             | HandleBits              | HandleClass |
|-------------|-------------------------|-------------|
|             | 5 4 3 2 1 0 9 8 7 ----- | 0           |
| Disk file   | N u u u u u u u         | 0           |
| Char device | N u u u u u u u         | 1           |
| Pipe        | N u u u u u u u         | 2           |

N is the NETWORK bit. If set, it means that the handle refers to a remote file, device, or pipe. Otherwise, the handle refers to a local file, device, or pipe.

u means that the bit is undefined and reserved. A program may not depend upon the values of these bits, as they may change in future versions of OS/2.

FlagWord is where the system returns the device's driver attribute word if HandType indicates a local character device.

**Returns:** IF ERROR (AX not = 0)

AX = Error Code:

- \* ERROR\_INVALID\_HANDLE - the handle is not in use or is attached to a physical device.

**Remarks** This function allows some programs which may be interactive or file-oriented to determine the source of their input. For example, CMD.EXE suppresses writing prompts if the input is from a disk file.



#### 2.1.8.6.35 DosQPathInfo - Query a file or a subdirectory for information:

Purpose Returns information for a specific file or subdirectory.

Format Calling Sequence:

EXTRN DosQPathInfo:FAR

```
PUSH@ ASCIIZ PathName ; File/directory name
PUSH WORD PathInfoLevel ; Data required
*
* PUSH@ OTHER PathInfoBuf ; Data buffer
* PUSH WORD PathInfoBufSize ; Data buffer size
PUSH DWORD 0 ; Reserved (must be zero)
CALL DosQPathInfo
```

Where PathName is the ASCIIZ full path name of the file or subdirectory. Meta characters are legal in the name only for PathInfoLevels five and six.

PathInfoLevel is the level of path information required.

Path information, where applicable, is based on the most recent DosClose, DosSetFileInfo, or DosSetPathInfo.

Level 0x0001 information is returned in the following format:

```
struct {
 unsigned short dateCreate;
 unsigned short timeCreate;
 unsigned short dateAccess;
 unsigned short timeAccess;
 unsigned short dateWrite;
 unsigned short timeWrite;
 long cbEOF;
 long cbAlloc;
 unsigned short attr;
};
```

Level 0x0002 simply adds cbList to level 0x0001, and uses the following format:

```
struct {
 unsigned short dateCreate;
 unsigned short timeCreate;
 unsigned short dateAccess;
 unsigned short timeAccess;
 unsigned short dateWrite;
 unsigned short timeWrite;
 long cbEOF;
 long cbAlloc;
 unsigned short attr;
 unsigned long cbList;
};
```

Level 0x0003 path information, returns a subset of the EA information for the file. On input, PathInfoBuf is an EAOP structure above. fpGEAList points to a GEA list defining the attribute names whose values will be returned. fpFEAList points to a data area where the relevant FEA list will be returned. The length field of this FEA list is valid, giving the size of the FEA list buffer. offError points to the offending GEA entry in case of error.

On output, PathInfoBuf is unchanged. The buffer pointed to by fpFEAList is filled in with the returned information.

If the buffer fpFEAList points to isn't large enough to hold the returned information (ie ERROR\_BUFFER\_OVERFLOW) cbList will still be valid, assuming there's at least enough space for it. Its value will be the size of the entire EA set for the file, even though only a subset of attributes was requested.

\*\*\*THE FOLLOWING UNPUBLISHED FUNCTION WILL GO AWAY IN FUTURE \*\*\*\*\*  
\*\*\*RELEASES AND SHOULD BE UNPUBLISHED\*\*\*

Level 0x0004 path information returns all EA info for the file. On input, PathInfoBuf is an EAOP structure above. fpGEAList contents are ignored. fpFEAList points to a data area where the relevant FEA list will be returned. The length field of this FEA list is valid, giving the size of the FEA list buffer. offError is ignored.

On output, PathInfoBuf is unchanged. The buffer pointed to by fpFEAList is filled in with the returned information.

It is important that applications expecting to be compatible with future versions of OS/2 not use this infolevel, since the limit on the size of the full EA list will be raised. Thus, this level is documented for utility and system use only.

+ If the buffer fpFEAList points to isn't large enough to hold the +  
+ returned information (ie ERROR\_BUFFER\_OVERFLOW) cbList will still  
+ be valid.

+ \*\*\*\*\*End of the UNPUBLISHED FUNCTION\*\*\*\*\*

| Level 0x0005 path information returns the fully qualified (true)  
| ASCIIZ name of PathName in PathInfoBuf. The pathname may contain  
| meta characters.

| Level 0x0006 requests a file system to verify the correctness of  
| PathName per its rules of syntax. An erroneous name is indicated  
| by an error return-code. The pathname may contain meta  
| characters.

\* PathInfoBuf is the storage area where the system returns the  
\* requested level of path information.

\* PathInfoBufSize is the length of PathInfoBuf.

Returns: IF ERROR (AX not = 0)

AX = Error Code:

- \* ERROR\_FILENAME\_EXCED\_RANGE - the path specified is unacceptable to the FSD managing the volume.
- \* ERROR\_PATH\_NOT\_FOUND - a component of PathName is not found.
- \* ERROR\_BUFFER\_OVERFLOW - the specified buffer is too short for the returned data.
- \* ERROR\_INVALID\_LEVEL - the specified PathInfoLevel is not supported.
- \* Device-driver/device-manager errors listed "-----  
-----" on page ---.
- \* ERROR\_INVALID\_EA\_NAME - there is an illegal character in an EA name. The offError field points to the offending GEA.
- + ERROR\_EA\_LIST\_INCONSISTENT - the cbList does not match the  
+ sum of the lengths of the GEA structures.
- + ERROR\_SHARING\_VIOLATION - the associated file or directory  
+ is being accessed.

+ Remarks: DosQPathInfo requires read/deny\_write sharing access for all  
+ infolevels, and will fail if some process holds conflicting

sharing rights to the file or directory.

\* 2.1.8.6.36 DosQSysInfo - Query System Information:

\* Purpose Query static system variables

\* Format Calling Sequence:

\* EXTRN DosQSysInfo:FAR

\* PUSH WORD Index ; Which variable  
\* PUSH@ OTHER DataBuf ; System info buffer  
\* PUSH WORD DataBufLen ; Data buffer size  
\* CALL DosQSysInfo

\* Where

\* Index is the ordinal of the system variable to return.

| \* Index == 0 indicates maximum path length. The maximum path  
| length will be returned in the first word of the DataBuf.

\* DataBuf is where the system returns the variable value.

\* DataBufLen is the length of the data buffer.

\* Returns: IF ERROR (AX not = 0)

\* AX = Error Code:

\* \* ERROR\_INVALID\_PARAMETER - the index is invalid.

\* \* ERROR\_BUFFER\_OVERFLOW - the specified buffer is too short  
\* for the returned data.

2.1.8.6.37 DosQVerify - Query Verify Setting:

Purpose Returns the value of the Verify flag.

Format Calling Sequence:

EXTRN DosQVerify:FAR

PUSH@ WORD VerifySetting ; Verify setting  
CALL DosQVerify

Where VerifySetting is the current Verify mode for the requesting process

\* If value = 0x00 is returned, verify mode is not active

\* If value = 0x01 is returned, verify mode is active

Returns: IF ERROR (AX not = 0)

AX = Error Code:

\* none

Remarks

#### 2.1.8.6.38 DosRead - Read from a File:

**Purpose** Reads the specified number of bytes from a file to a buffer location.

**Format** Calling Sequence:

EXTRN DosRead:FAR

```
PUSH WORD FileHandle ; File Handle
PUSH@ OTHER BufferArea ; Address of user buffer
PUSH WORD BufferLength ; Buffer length
PUSH@ WORD BytesRead ; Bytes read
CALL DosRead
```

**Where** FileHandle is the 2-byte file handle obtained from DosOpen.

BufferArea is address of the input buffer.

BufferLength is the number of bytes to be read.

BytesRead is where the system returns the number of bytes read.

**Returns:** IF ERROR (AX not = 0)

AX - Error Code:

- ERROR\_INVALID\_HANDLE - the handle is not in use or is attached to a physical device.
- ERROR\_ACCESS\_DENIED - the handle is opened as write-only.
- ERROR\_LOCK\_VIOLATION - the region of the file is partially or completely locked by another process.
- Device-driver/device-manager errors listed "-----" on page ---.

**Remarks** It is not guaranteed that the requested number of bytes will be read.

A BufferLength value of zero is not considered an error. In the case when the value of BufferLength is zero, the system treats the request as a null operation.

**Named Pipe Considerations** Reads bytes or messages from a pipe. There are three cases:

1. Byte pipe. The pipe must be in byte read mode (an error is

returned if in message read mode). All currently available data, up to the size requested, is returned.

2. Message pipe, message read mode. A read that is larger than the next available message returns only that message and BytesRead is set to indicate the size of the message returned. A read that is smaller than the next available message returns with the number of bytes requested and a MORE DATA error code. Subsequent DosReads will continue reading the message. DosPeekNmPipe may be used to determine how many bytes are left in the message.

3. Message pipe, byte read mode. Reads the pipe as if it were a byte stream, skipping over message headers. This is like reading a byte pipe in byte mode.

When blocking mode is set, a read blocks until data is available. In this case, the read will never return with BytesRead=0 except at EOF. Note that in message read mode, messages are always read in their ENTIRETY, except in the case where the message is bigger than the size of the read.

When nonblocking mode is set, a read will return with BytesRead=0 at EOF. An error will be returned if there is no data available.

Note: when resuming the reading of a message after a MORE DATA indication, the reads will always block until the next piece (or rest) of the message can be transferred. Non-blocking mode only allows a return with BytesRead=0 when trying to read at the start of a message and no message is available.

#### 2.1.8.6.38 DosRead - Read from a File:

**Purpose** Reads the specified number of bytes from a file to a buffer location.

**Format** Calling Sequence:

EXTRN DosRead:FAR

```
PUSH WORD FileHandle ; File Handle
PUSH@ OTHER BufferArea ; Address of user buffer
PUSH WORD BufferLength ; Buffer length
PUSH@ WORD BytesRead ; Bytes read
CALL DosRead
```

**Where** FileHandle is the 2-byte file handle obtained from DosOpen.

BufferArea is address of the input buffer.

BufferLength is the number of bytes to be read.

BytesRead is where the system returns the number of bytes read.

**Returns:** IF ERROR (AX not = 0)

AX = Error Code:

- \* ERROR\_INVALID\_HANDLE - the handle is not in use or is attached to a physical device.
- \* ERROR\_ACCESS\_DENIED - the handle is opened as write-only.
- \* ERROR\_LOCK\_VIOLATION - the region of the file is partially or completely locked by another process.
- \* Device-driver/device-manager errors listed "-----  
-----" on page ---.

**Remarks** It is not guaranteed that the requested number of bytes will be read.

A BufferLength value of zero is not considered an error. In the case when the value of BufferLength is zero, the system treats the request as a null operation.

**Named Pipe Considerations** Reads bytes or messages from a pipe. There are three cases:

1. Byte pipe. The pipe must be in byte read mode (an error is

returned if in message read mode). All currently available data, up to the size requested, is returned.

2. Message pipe, message read mode. A read that is larger than the next available message returns only that message and BytesRead is set to indicate the size of the message returned. A read that is smaller than the next available message returns with the number of bytes requested and a MORE DATA error code. Subsequent DosReads will continue reading the message. DosPeekNmPipe may be used to determine how many bytes are left in the message.
3. Message pipe, byte read mode. Reads the pipe as if it were a byte stream, skipping over message headers. This is like reading a byte pipe in byte mode.

When blocking mode is set, a read blocks until data is available. In this case, the read will never return with BytesRead=0 except at EOF. Note that in message read mode, messages are always read in their ENTIRETY, except in the case where the message is bigger than the size of the read.

When nonblocking mode is set, a read will return with BytesRead=0 at EOF. An error will be returned if there is no data available.

Note: when resuming the reading of a message after a MORE DATA indication, the reads will always block until the next piece (or rest) of the message can be transferred. Non-blocking mode only allows a return with BytesRead=0 when trying to read at the start of a message and no message is available.

#### 2.1.8.6.39 DosReadAsync - Async Read from a File:

**Purpose** Transfers the specified number of bytes from a handle to a buffer location, asynchronously with respect to the requesting process's execution.

**Format** Calling Sequence:

EXTRN DosReadAsync:FAR

```
PUSH WORD FileHandle ; File handle
PUSH@ DWORD RamSemaphore ; Address of Ram semaphore
PUSH@ WORD ReturnCode ; Address of I/O error RC
PUSH@ OTHER BufferArea ; Address of user buffer
PUSH WORD BufferLength ; Buffer length
PUSH@ WORD BytesRead ; Bytes read
CALL DosReadAsync
```

**Where** FileHandle is the 2-byte file handle obtained from DosOpen.

RamSemaphore is used by the system to post operation complete to the caller.

ReturnCode is where the system returns the I/O operation return code.

BufferArea is address of the input buffer.

BufferLength is the number of bytes to be read.

BytesRead is where the system returns the number of bytes read.

**Returns:** AX = 0 ReturnCode = Error code if error

AX = Error Code:

- \* ERROR\_INVALID\_HANDLE - the handle is not in use or is attached to a physical device.
- \* ERROR\_ACCESS\_DENIED - the handle is opened as write-only.
- \* ERROR\_LOCK\_VIOLATION - the region of the file is partially or completely locked by another process.
- \* Device-driver/device-manager errors listed "-----" on page ---.

**Remarks** It is not guaranteed that the requested number of bytes will be read.

A BufferLength value of zero is not considered an error. In the case when the value of BufferLength is zero, the system treats the request as a null operation.

RamSemaphore must be set by the application before the DosReadAsync call is made. The application issues the following sequence:

```
* DosSemSet ...
* DosReadAsync ...
* DosSemWait ...
```

**Named Pipe Considerations** Reads bytes or messages from a pipe. There are three cases:

1. Byte pipe. The pipe must be in byte read mode (an error is returned if in message read mode). All currently available data, up to the size requested, is returned.
2. Message pipe, message read mode. A read that is larger than the next available message returns only that message and BytesRead is set to indicate the size of the message returned. A read that is smaller than the next available message returns with the number of bytes requested and a MORE DATA error code. Subsequent DosReads will continue reading the message. DosPeekNmPipe may be used to determine how many bytes are left in the message.
3. Message pipe, byte read mode. Reads the pipe as if it were a byte stream, skipping over message headers. This is like reading a byte pipe in byte mode.

When blocking mode is set, a read blocks until data is available. In this case, the read will never return with BytesRead=0 except at EOF. Note that in message read mode, messages are always read in their ENTIRETY, except in the case where the message is bigger than the size of the read.

When nonblocking mode is set, a read will return with BytesRead=0 if no data is available at the time of the read.

**Note:** when resuming the reading of a message after a MORE DATA indication, the reads will always block until the next piece (or rest) of the message can be transferred. Non-blocking mode only allows a return with BytesRead=0 when trying to read at the start of a message and no message is available.

#### 2.1.8.6.40 DosRmdir - Remove Subdirectory:

**Purpose** Removes a subdirectory from the specified disk.

**Format** Calling Sequence:

EXTRN DosRmdir:FAR

PUSH@ ASCIIZ DirName ; Directory name  
PUSH DWORD 0 ; Reserved (must be zero)  
CALL DosRmdir

**Where** DirName is the ASCIIZ directory path name.

**Returns:** IF ERROR (AX not = 0)

AX = Error Code:

- \* ERROR\_FILENAME\_EXCED\_RANGE - the path specified is unacceptable to the FSD managing the volume.
- \* ERROR\_PATH\_NOT\_FOUND - the drive is invalid or a directory in the path is not found.
- \* ERROR\_CURRENT\_DIRECTORY - the specified path is the current directory of a process.
- \* ERROR\_ACCESS\_DENIED - the root directory is specified, or the directory is not empty.
- \* Device-driver/device-manager errors listed "-----  
-----" on page ---.

**Remarks** The directory must be empty before it can be removed with the exception of the "." and "..". You cannot remove subdirectories that contain hidden files. The last directory name in the path is the directory to be removed. The root directory and the current directory cannot be removed.

] DosQSysInfo must be used by an application to determine the  
] maximum path length supported by OS/2. The returned value should  
] be used to dynamically allocate buffers that are to be used to  
] store paths. This will ensure that applications function  
] correctly (wrt path lengths) for future versions of OS/2. The  
] path length includes the drive specifier (i.e. d:), the leading  
] '\', and the trailing NUL.

#### 2.1.8.6.41 DosScanEnv - Scan Environment Segment:

**Purpose** Scans an environment segment for an environment variable.

**Format** Calling Sequence:

```
EXTRN DosScanEnv:FAR
```

```
PUSH@ ASCIIZ EnvVarName ; Environment variable name
PUSH@ DWORD ResultPointer ; Search result pointer
CALL DosScanEnv
```

**Where** EnvVarName points to the ASCIIZ name of the environment variable of interest. Do not include a trailing "=", since this is not part of the name.

ResultPointer is where the system returns the pointer to the environment string. ResultPointer points to the first character of the string which is the value of the environment variable and can be passed directly into DosSearchPath.

**Returns:** IF ERROR (AX not = 0)

AX = Error Code:

- \* ERROR\_ENVVAR\_NOT\_FOUND - the specified environment variable is not found in the environment segment.

**Remarks** Assume that the process's environment contains:

```
"DPATH=c:\sysdir;c:\libdir"
A
```

```
|
```

```
+---- ResultPointer points here after call
 to DosScanEnv below.
```

```
DosScanEnv("DPATH", &ResultPointer);
```

As noted above, ResultPointer will point to the first character of the value of the environment variable.

#### 2.1.8.6.42 DosSearchPath - Search a path for a file name:

**Purpose** Provides a general path search mechanism which allows applications to find files residing along paths. The path string may come from the process's Environment, or be supplied directly by the caller.

**Format** Calling Sequence:

```
EXTRN DosSearchPath:FAR
```

```
PUSH WORD Control ; Function control vector
PUSH@ ASCIIZ PathRef ; Search path reference
PUSH@ ASCIIZ FileName ; File name
PUSH@ OTHER ResultBuffer ; Search result buffer
PUSH WORD ResultBufferLen ; Search result buffer length
CALL DosSearchPath
```

**Where**

Control is a word bit vector which controls the behaviour of DosSearchPath

- \* Bit 0 = Implied Current Bit
- \* Bit 1 = Path Source Bit
- \* Bit 2 = Ignore Network Errors Bit
- \* Bits 3-15 = Reserved bits, must be 0.

The Implied Current Bit controls whether the current directory is implicitly on the front of the search path. If the Implied Current Bit = 0, DosSearchPath will only search the current directory if it appears in the search path. If the Implied Current Bit = 1, DosSearchPath will search the current working directory before it searches the directories in the search path.

Implied Current Bit = 0 and Path = ".;a;b"

is equivalent to

Implied Current Bit = 1 and Path = "a;b"

The Path Source Bit determines how DosSearchPath interprets the PathRef argument. If the Path Source Bit = 0, then PathRef points to the actual search path. The search path string may be anywhere in the calling process's address space, hence it may be in the environment, but does not have to be.



If the Path Source Bit = 1, then PathRef points to the name of an environment variable in the process's environment, and that environment variable contains the search path.

PathRef points to an ASCIIZ string.

If the Path Source Bit of Control = 0, then PathRef points directly to the search path, which may be anywhere in the caller's address space.

If the Path Source Bit of Control = 1, then PathRef points to a string which is the name of an environment variable which contains the search path.

A search path consists of a sequence of paths separated by ";". It is a single ASCIIZ string. The directories will be searched in the order they appear in the path.

Environment variable names are simply strings which match name strings in the environment. The "=" sign is not part of the name.

The Ignore Network Errors Bit controls whether the search will abort if it encounters a network error or will continue the search with the next element. This allows one to place network paths in the PATH variable and be able to find executables in components of the PATH variable even if the network returns an error, e.g. if a server is down. If the Ignore Network Errors Bit = 0, DosSearchPath will abort the search if it encounters an error from the network. If the Ignore Network Errors Bit = 1, DosSearchPath will continue on the search if it encounters network errors.

FileName points to the ASCIIZ file name to search for. It may contain wild cards. If FileName does contain wild cards, they will remain in the result path returned in

ResultBuffer. This allows applications like cmd.exe to feed the output directly to DosFindFirst. If there are no wildcards in FileName, the result path returned in ResultBuffer will be a full qualified name, and may be passed directly to DosOpen, or any other system call.

ResultBuffer holds the result pathname of the file, if found. If FileName is found in one of the directories along the path, its full pathname will be returned in ResultBuffer. (With wildcards from FileName left in place.) Do not depend on the contents of ResultBuffer being meaningful if DosSearchPath doesn't return with AX=0.

ResultBufLen is the length of ResultBuffer, in bytes.

Returns: IF ERROR (AX not = 0)

AX = Error Code:

- \* ERROR\_BUFFER\_OVERFLOW - the specified buffer is too short for the returned data.
- \* ERROR\_FILE\_NOT\_FOUND - no matching file is found.
- \* Device-driver/device-manager errors listed "-----" on page ---.

Remarks PathRef always points to an ASCIIZ string.

example:

Let DPATH be an environment variable in the environment segment of the process.

"DPATH=c:\sysdir;c:\init" /\* in the environment \*/

The following two code fragments are equivalent:

```
DosScanEnv("DPATH", &PathRef);
DosSearchPath(0, /* Path Source Bit = 0 */
PathRef, "myprog.ini", &ResultBuffer, ResultBufLen);
```

```
DosSearchPath(2, /* Path Source Bit = 1 */
"DPATH", "myprog.ini", &ResultBuffer, ResultBufLen);
```

Both of them use the search path stored as DPATH in the environment segment. In the first case, the application uses DosScanEnv to find the variable, in the second case DosSearchPath calls DosScanEnv for the application.

DosSearchPath does no consistency checking or formatting on the names, it simply does a DosFindFirst on a series of names it constructs from PathRef and FileName. This means that the underlying file system does the name formatting.

DosQSysInfo must be used by an application to determine the maximum path length supported by OS/2. The returned value should be used to dynamically allocate buffers that are to be used to store paths. This will ensure that applications function correctly (wrt path lengths) for future versions of OS/2. The path length includes the drive specifier (i.e. d:), the leading '\', and the trailing NUL.

#### 2.1.8.6.43 DosSelectDisk - Select Default Drive:

**Purpose** Selects the drive specified as the default drive for the calling process.

**Format** Calling Sequence:

```
EXTRN DosSelectDisk:FAR
```

```
PUSH WORD DriveNumber ; Default drive number
CALL DosSelectDisk
```

**Where**

DriveNumber contains the default drive number (1=A, 2=B, ... ).

**Returns:** IF ERROR (AX not = 0)

AX = Error Code:

\* ERROR\_INVALID\_DRIVE - the specified drive is invalid.

#### 2.1.8.6.44 DosSetFHandState - Set File Handle State:

**Purpose** Set the state of the specified file.

**Format** Calling Sequence:

```
EXTRN DosSetFHandState:FAR
```

```
PUSH WORD FileHandle ; File handle
PUSH WORD FileHandleState ; File handle state
CALL DosSetFHandState
```

**Where**

FileHandle is the handle of the file to be set.

FileHandleState is the file handle state and consists of the following bit fields. They are the:

- \* Inheritance flag
- \* Write-through flag
- \* Fail-errors flag
- \* Zero bit field

##### File Handle State Bits

```
5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
0 W F C R R R R I 0 0 0 R 0 0 0
```

**I** Inheritance Flag

If I = 0; File handle is inherited by a spawned process resulting from a DosExecPgm call.

If I = 1; File handle is private to the current process.

This bit is not inherited by child processes.

**W** File Write-through

The file is opened as follows:

If W = 0; Writes to the file may be run through the DOS buffer cache.

If W = 1; Writes to the file may go through the DOS buffer cache but, the sectors will be written (actual file I/O completed) before a synchronous write call returns. This state of the file defines it as a synchronous file.

NOTE that this is a mandatory bit in that the data MUST be written out to the medium for synchronous write operations. This bit is not inherited by child processes.

F Fail-Errors

Media I/O errors will be handled as follows:

If F = 0; reported through the system critical error handler.  
If F = 1; reported directly to the caller via return code.

This bit is not inherited by child processes. Media I/O errors generated through an Ioctl Category 8 function, always get reported directly to the caller via return code. The Fail-Errors functionality applies only to non-Ioctl handle-based type file I/O calls.

C Cache/No-Cache

The file is opened as follows:  
If C = 0; I/O to the file need not be done through the disk driver cache.  
If C = 1; It is advisable for the disk driver to cache the data in I/O operations on this file.

This bit is an ADVISORY bit, and is used to advise FSDs and device drivers on whether it is worth caching the data or not. This bit, like the write-through bit, is a per-handle bit. It is not inherited by child processes.

0 Zero bits

These bits must be set to zero.

Any other values for FileHandleState are invalid.

R Reserved bits

These bits are reserved and should be set to the values returned by DosQFHandState in these positions.

Returns: IF ERROR (AX not = 0)

AX = Error Code:

- \* ERROR\_INVALID\_HANDLE - the handle is not in use or is attached to a physical device.
- \* ERROR\_INVALID\_PARAMETER - the specified mode contains reserved fields non-zero or contains an invalid value.

Remarks Setting the write-through flag does not affect any previous writes which may have been done. That data may remain in the buffers.

The file handle state bits set by this function can be queried with the DosQFHandState function call.

Named Pipe Considerations Allows setting of the inheritance (I) and write-through (W) bits. Note that setting W to 1 prevents write-behind operations on remote pipes.

#### 2.1.8.6.45 DosSetFileInfo - Set a File's Information:

**Purpose** Specifies information for a file.

**Format** Calling Sequence:

EXTRN DosSetFileInfo:FAR

```
PUSH WORD FileHandle ; File handle
PUSH WORD FileInfoLevel ; File info data type
PUSH@ OTHER FileInfoBuf ; File info buffer
PUSH WORD FileInfoBufSize ; File info buffer size
CALL DosSetFileInfo
```

**Where** FileHandle is the file handle.

FileInfoLevel is the level of file information being defined.

Level 0x0001 file information is set from FileInfoBuf in the following format:

```
struct {
 unsigned short dateCreate;
 unsigned short timeCreate;
 unsigned short dateAccess;
 unsigned short timeAccess;
 unsigned short dateWrite;
 unsigned short timeWrite;
};
```

Level 0x0002 file information sets a series of EA name/value pairs. On input, FileInfoBuf is an EAOP structure above. fpGEAList is ignored. fpFEAList points to a data area where the relevant FEA list is to be found. offError is ignored.

On output, fpGEAList is unchanged. fpFEAList is unchanged as is the area pointed to by fpFEAList. If an error occurred during the set, offError will be the offset of the FEA where the error occurred. The API return code will be the error code corresponding to the condition generating the error. If no error occurred, offError is undefined.

FileInfoBuf is the storage area where the system gets the file information.

FileInfoBufSize is the length of FileInfoBuf.

**Returns:** IF ERROR (AX not = 0)

**AX** = Error Code:

- \* ERROR\_INVALID\_LEVEL - the specified FileInfoLevel is not supported.
- \* ERROR\_INSUFFICIENT\_BUFFER - the specified buffer is too short to contain the stated level of information.
- \* ERROR\_DIRECT\_ACCESS\_HANDLE - the handle is opened with the direct-open bit set.
- \* ERROR\_INVALID\_PARAMETER - invalid times/dates are specified. This may be returned by an FSD if it validates the date and time.
- \* ERROR\_INVALID\_EA\_NAME - there is an illegal character in an EA name. The offError field points to the offending FEA.
- \* ERROR\_EA\_LIST\_INCONSISTENT - the cbList does not match the sum of the lengths of the FEA structures.
- \* ERROR\_EA\_VALUE\_UNSUPPORTED - the FSD detects an error in the value portion of an EA. The offError field points to the offending FEA.
- \* ERROR\_ACCESS\_DENIED - the specified file is not open for write access.
- \* Device-driver/device-manager errors listed "-----" on page ---.

\* **Remarks** The DosSetFileInfo level 0x0001 structure is a prefix of the DosQFileInfo level 0x0001 structure.

DosSetFileInfo will work only for files opened in a mode that allows write-access.

A zero (0) value in both the date and time components of a field will cause that field to be left unchanged. For example, if both 'Last write date' and 'Last write time' are specified as zero in the Level 0x0001 information structure, then both attributes of the file are left unchanged. If either 'Last write date' or 'Last write time' are specified as non-zero, then both attributes of the file will be set to the new values.

The FAT file system supports only the modification date and time. Creation and access dates and times will not be affected.

#### 2.1.8.6.46 DosSetFileMode - Set File Mode:

**Purpose** Change the mode (attribute) of the specified file.

**Format** Calling Sequence:

EXTRN DosSetFileMode:FAR

```
PUSH@ ASCIIZ FileName ; File path name
PUSH WORD NewAttribute ; New attribute of file
PUSH DWORD 0 ; Reserved (must be zero)
CALL DosSetFileMode
```

**Where** FileName is the ASCIIZ file path name.

NewAttribute is the file's new attribute.

**Returns:** IF ERROR (AX not = 0)

AX = Error Code:

- \* ERROR\_PATH\_NOT\_FOUND - a component of the directory path is not present.
- \* ERROR\_FILE\_NOT\_FOUND - the last component of the file name is not present on the media.
- \* ERROR\_FILENAME\_EXCED\_RANGE - the path specified is unacceptable to the FSD managing the volume.
- \* ERROR\_ACCESS\_DENIED - the attributes are invalid or the specified file name is a character device or a root directory was specified.
- \* ERROR\_SHARING\_BUFFER\_EXCEEDED - there is not enough memory to hold sharing information.
- \* ERROR\_SHARING\_VIOLATION - the file is currently in use by another process.

Device-driver/device-manager errors listed "-----" on page ---.

**Remarks** Attributes for Volume Label (0x0008) or Subdirectory (0x0010) cannot be set using DosSetFileMode. If the above referenced attributes are used to change a file's mode, an error code is returned. Attributes of root directories cannot be changed and an error will be returned.

File attributes are defined as follows.

0x0001 read only file.

0x0002 hidden file.

0x0004 system file.

0x0008 volume label.

0x0010 subdirectory.

0x0020 file archive.

All other attributes are reserved.

DosQSysInfo must be used by an application to determine the maximum path length supported by OS/2. The returned value should be used to dynamically allocate buffers that are to be used to store paths. This will ensure that applications function correctly (wrt path lengths) for future versions of OS/2. The path length includes the drive specifier (i.e. d:), the leading '\', and the trailing NUL.

#### 2.1.8.6.47 DosSetFsInfo - Set File System Information:

**Purpose** Set information for a file system device.

**Format** Calling Sequence:

EXTRN DosSetFsInfo:FAR

```
PUSH WORD DriveNumber ; Drive number
PUSH WORD FSInfoLevel ; File system data type
PUSH@ OTHER FSInfoBuf ; File system info buffer
PUSH WORD FSInfoBufSize ; File system info buffer size
CALL DosSetFsInfo
```

**Where** DriveNumber is the logical drive number (0=default, 1=A, 2=B, 3=C, ... ) and represents the FSD for the media currently in that drive. A value of '0xFFFF' notes that FSInfoBuf contains the ASCIIIZ path name of the FSD.

\* FSInfoLevel is the level of file information to set.

Level 0x0001 is reserved.

Level 0x0002 information is defined in the following format:

```
struct {
 unsigned char cbVolLabel;
 unsigned char szVolLabel[];
};
```

cbVolLabel length of volume label

szVolLabel asciiz volume label.

FSInfoBuf is where the system gets the new file system information.

FSInfoBufSize is the length of FSInfoBuf.

**Returns:** IF ERROR (AX not = 0)

AX = Error Code:

- \* ERROR\_INVALID\_LEVEL - the specified FSInfoLevel is not supported.
- \* ERROR\_INVALID\_DRIVE - the specified drive is invalid.
- \* ERROR\_CANNOT\_MAKE - cannot create volume label

- \* ERROR\_INVALID\_NAME - there are characters in the volume label that are illegal.
- \* ERROR\_INSUFFICIENT\_BUFFER - the specified buffer is too short to contain the stated level of information.
- \* ERROR\_LABEL\_TOO\_LONG - the volume label exceeded the file systems name capacity.
- \* Device-driver/device-manager errors listed "-----  
-----" on page ---.

#### Remarks

Trailing blanks supplied at volume label definition time are not returned by DosQFsInfo.

#### 2.1.8.6.48 DosSetMaxFH - Set Maximum File Handles:

**Purpose** This function call defines the maximum number of file handles for the current process.

**Format** Calling Sequence:

```
EXTRN DosSetMaxFH:FAR
```

```
PUSH WORD NumberHandles ; Number of file handles
CALL DosSetMaxFH
```

**Where**

NumberHandles is the total number of file handles to be provided.

**Returns:** IF ERROR (AX not = 0)

AX = Error Code:

- \* ERROR\_NOT\_ENOUGH\_MEMORY - unable to allocate storage for new file handle table.
- \* ERROR\_INVALID\_PARAMETER - the requested number of file handles is > 32768 or the requested number is smaller than the currently allocated set.

**Remarks** All currently open file handles are preserved.

#### 2.1.8.6.49 DosSetPathInfo - Set a File's or Directory's Information:

**Purpose** Specifies information for a file or a directory.

**Format** Calling Sequence:

```
EXTRN DosSetPathInfo:FAR
```

```
PUSH# ASCIIZ PathName ; File/dir full name
PUSH WORD PathInfoLevel ; Info data type
PUSH# OTHER PathInfoBuf ; Info buffer
PUSH WORD PathInfoBufSize ; Info buffer size
PUSH WORD PathInfoFlags ; Flags
PUSH DWORD 0 ; Reserved (must be zero)
CALL DosSetPathInfo
```

**Where**

PathName is the ASCIIZ full path name of the file or subdirectory. This name may not contain meta characters.

PathInfoLevel is the level of file/directory information being defined.

Level 0x0001 file information is in the following format:

```
struct {
 unsigned short dateCreate;
 unsigned short timeCreate;
 unsigned short dateAccess;
 unsigned short timeAccess;
 unsigned short dateWrite;
 unsigned short timeWrite;
};
```

Level 0x0002 file information sets a series of EA name/value pairs. On input, PathInfoBuf is an EAOP structure above. fpGEAList is ignored. fpFEAList points to a data area where the relevant FEA list is to be found. offError is ignored.

On output, fpGEAList is unchanged. fpFEAList is unchanged as is the area pointed to by fpFEAList. If an error occurred during the set, offError will be the offset of the FEA where the error occurred. The API return code will be the error code corresponding to the condition generating the error. If no error occurred, offError is undefined.

PathInfoBuf is the storage area where the system gets the file information.

PathInfoBufSize is the length of PathInfoBuf.

PathInfoFlags contain information on how the Set operation is to be performed. Currently, only one bit is defined. If PathInfoFlags == 0x0010, then this indicates that the information being set must be written out to disk before returning to the application. This guarantees that the EAs have been written to disk. All other bits are reserved and must be zero.

Returns: IF ERROR (AX not = 0)

AX = Error Code:

- \* ERROR\_INVALID\_LEVEL - the specified PathInfoLevel is not supported.
- \* ERROR\_INVALID\_PARAMETER - invalid times/dates are specified.  
This may be returned by an FSD if it validates the date and time.
- \* ERROR\_FILENAME\_EXCED\_RANGE - the path specified is unacceptable to the FSD managing the volume.
- \* ERROR\_INSUFFICIENT\_BUFFER - the specified buffer is too short to contain the stated level of information.
- \* ERROR\_INVALID\_EA\_NAME - there is an illegal character in an EA name. The offError field points to the offending FEA.
- \* ERROR\_EA\_LIST\_INCONSISTENT - the cbList does not match the sum of the lengths of the FEA structures.
- \* ERROR\_EA\_VALUE\_UNSupportable - the FSD detects an error in the value portion of an EA. The offError field points to the offending FEA.
- \* ERROR\_SHARING\_VIOLATION - the associated file or directory is being accessed.

Device-driver/device-manager errors listed "-----" on page ---.

Remarks: DosSetPathInfo requires read/deny\_write sharing access for all infolevels, and will fail if some process holds conflicting sharing rights to the file or directory.

A zero (0) value in both the date and time components of a field will cause that field to be left unchanged. For example, if both 'Last write date' and 'Last write time' are specified as zero in the Level 0x0001 information structure, then both attributes of

the file are left unchanged. If either 'Last write date' or 'Last write time' are specified as non-zero, then both attributes of the file will be set to the new values.

The write-through bit in PathInfoFlags should be used only in cases where it is necessary, for data integrity purposes, to write the EAs out to disk immediately, instead of caching them and writing them out later. Setting the write-through bit all the time may degrade the performance.



#### 2.1.8.6.50 DosSetVerify - Set/Reset Verify Switch:

**Purpose** Sets the verify switch.

**Format** Calling Sequence:

EXTRN DosSetVerify:FAR

PUSH WORD VerifySetting ; New value of verify switch  
CALL DosSetVerify

**Where**

VerifySetting is the new state of Verify Mode for the requesting process

\* If value = 0 is specified, Verify Mode is deactivated

\* If value = 1 is specified, Verify Mode is activated

**Returns:** IF ERROR (AX not = 0)

AX = Error Code:

\* ERROR\_INVALID\_VERIFY\_SWITCH

**Remarks** When verify is on, device drivers are requested to perform a verify-after write operation each time they do a file write to assure proper data recording on the disk.

#### + 2.1.8.6.51 DosShutdown - Shutdown File Systems for Power Off:

+ **Purpose** Locks out changes to all file systems. Forces system buffers to disk in preparation for system power off.

+ **Format** Calling Sequence:

+ EXTRN DosShutdown:FAR

+ PUSH DWORD Reserved ; Reserved Zero  
+ CALL DosShutdown

+ **Where** Reserved is a dword whose value must be zero.

+ **Returns:** IF ERROR (AX not = 0)

+ AX = Error Code:

+ \* ERROR\_IN\_PROGRESS - a shutdown is already in progress.

+ \* ERROR\_CANNOT\_SHUTDOWN - the system was unable to shutdown one or more file systems.

+ \* ERROR\_INVALID\_PARMS - the reserved word was non-zero.

+ **Remarks** This function may take several minutes to complete depending on the amount of data buffered.

+ APIs that change file system data called while the system is shutdown will either return the error ERROR\_SYSTEM\_SHUTDOWN or block permanently.

+ It should be noted that it will not be possible to increase memory overcommit once the DosShutdown has been called. This means that in low memory situations some functions may fail due to a lack of memory. This is of particular importance to the process calling DosShutdown. All memory that the calling process will ever need should be allocated before calling DosShutdown. This includes implicit memory allocation that may be done on behalf of the caller by system functions.

+ When this api returns successfully, it is safe to power the system off or to reboot it.

#### 2.1.8.6.52 DosWrite - Synchronous Write to a File:

**Purpose** Transfers the specified number of bytes from a buffer to the specified file, synchronously with respect to the requesting process's execution.

**Format** Calling Sequence:

EXTRN DosWrite:FAR

```
PUSH WORD FileHandle ; File handle
PUSH# OTHER BufferArea ; Address of user buffer
PUSH WORD BufferLength ; Buffer length
PUSH# WORD BytesWritten ; Bytes written
CALL DosWrite
```

**Where** FileHandle is the 2-byte file handle obtained from DosOpen.

BufferArea is address of the output buffer.

BufferLength is the number of bytes to be written.

BytesWritten is where the system returns the number of bytes written.

**Returns:** IF ERROR (AX not = 0)

AX = Error Code:

- \* ERROR\_INVALID\_HANDLE - the handle is not in use or is attached to a physical device.
- \* ERROR\_LOCK\_VIOLATION - the region of the file is partially or completely locked by another process.
- \* ERROR\_ACCESS\_DENIED - the file is opened as read-only.
- \* Device-driver/device-manager errors listed "-----  
-----" on page ---.

#### Remarks

Upon return from this function, BytesWritten is the number of bytes actually written.

A BufferLength value of zero is not considered an error. In the case when the value of BufferLength is zero, the system treats the request as a null operation.

**Named Pipe Considerations** Writes bytes or messages to a pipe. Each write to a message pipe writes a message whose size is the length of the write; DosWrite automatically encodes message lengths in the pipe, so applications need not encode this information in the buffer being written.

Writes in blocking mode always write all requested bytes before returning. In nonblocking mode, writes return either with all bytes written or none written; the latter will occur in certain cases where the DosWrite would have to block in order to complete the request (e.g., no room in pipe buffer or buffer currently being written by another client).

An attempt to write to a pipe whose other end has been closed will return with ERROR\_BROKEN\_PIPE.

#### 2.1.8.6.53 DosWriteAsync - Asynchronous Write to a File:

**Purpose** Transfers the specified number of bytes to a handle from a buffer location, asynchronously with respect to the requesting process's execution.

**Format** Calling Sequence:

EXTRN DosWriteAsync:FAR

```
PUSH WORD FileHandle ; File handle
PUSH@ DWORD RamSemaphore ; Address of Ram semaphore
PUSH@ WORD ReturnCode ; Address of I/O error RC
PUSH@ OTHER BufferArea ; Address of user buffer
PUSH WORD BufferLength ; Buffer length
PUSH@ WORD BytesWritten ; Bytes written
CALL DosWriteAsync
```

**Where** FileHandle is the 2-byte file handle obtained from DosOpen.

RamSemaphore is used by the system to post operation complete to the caller.

ReturnCode is where the system returns the I/O operation return code.

BufferArea is address of the output buffer.

BufferLength is the number of bytes to be written.

BytesWritten is where the system returns the number of bytes written.

**Returns:** AX = 0 ReturnCode = Error code if error

##### Error Codes:

- \* ERROR\_INVALID\_HANDLE - the handle is not in use or is attached to a physical device.
- \* ERROR\_LOCK\_VIOLATION - the region of the file is partially or completely locked by another process.
- \* ERROR\_ACCESS\_DENIED - the file is opened as read-only.
- \* Device-driver/device-manager errors listed "-----  
-----" on page ---.

**Remarks**

Upon return from this function, BytesWritten is the number of bytes actually written.

A BufferLength value of zero is not considered an error. In the case when the value of BufferLength is zero, the system treats the request as a null operation.

RamSemaphore must be set by the application before the DosWriteAsync call is made. The application issues the following sequence:

- \* DosSemSet ...
- \* DosWriteAsync ...
- \* DosSemWait ...

**Named Pipe Considerations** Writes bytes or messages to a pipe. Each write to a message pipe writes a message whose size is the length of the write; DosWrite automatically encodes message lengths in the pipe, so applications need not encode this information in the buffer being written.

Writes in blocking mode always write all requested bytes before returning. In nonblocking mode, writes return either with all bytes written or none written; the latter will occur in certain cases where the DosWrite would have to block in order to complete the request (e.g., no room in pipe buffer or buffer currently being written by another client).

An attempt to write to a pipe whose other end has been closed will return with ERROR\_BROKEN\_PIPE.

#### 2.1.8.7 FSD System Interfaces

#### 2.1.8.8 Overview of Installable File System Driver Dispatch

##### Notes:

Installable file system entry points are called by the kernel as a result of action taken through the published standard file I/O application programming

interface in the 1.2 version of the OS/2 Operating System and Utilities Functional Specification.

Installable file systems will be installed as OS/2 dynamic link library modules. Unlike device drivers, they may include any number of segments, all of which will remain after initialization unless the FSD itself takes some action to free them.

An FSD will export FS entries to the kernel using standard public declarations. Each FS entry will be called directly. The kernel will manage the association between internal data structures and FSDs.

When a file system service is required, OS/2 will assemble an argument list, and call the appropriate FS entry for the relevant FSD. If a back-level FSD is loaded, the kernel will assure that all arguments passed and all structures passed are understood by the FSD.

Application program interfaces that are unsupported by an FSD, must receive ERROR\_UNSUPPORTED\_FUNCTION from the FSD.

Certain routines, i.e., FS\_PROCESSNAME, may provide no processing, because 1) no processing is needed, or 2) processing does not make sense. The routines should return no error, not ERROR\_NOT\_SUPPORTED.

#### 2.1.8.9 Data Structures

As mentioned previously, OS/2 data structures will need to be expanded to include a pointer to the file system driver. The affected structures include the CDS (current directory structure), the SFT (system file table entry), the VPB (volume parameter block), and the file search structures. In addition, these structures need to include areas which are defined to be file system specific.

The file system service routines will generally be passed pointers to two parameter areas in addition to read-only parameters which will be specific to each call. The FSD does not need to verify these pointers. The two parameter areas will contain file system independent data which are maintained jointly by the OS/2 and the file system driver and an area of file system dependent data which will be unused by OS/2 and which may be used in any way the file system driver wishes. The file system driver is generally permitted to use the file system dependent information in any way it sees fit; it may contain all the information needed to describe the current state of the file or directory, or it may contain a 'handle' which will direct it to other information about the file maintained within the FSD. Any handles must be GDT selectors as any SFT, CDS, or VPB may be seen by more than one process.

The following definitions are used in the file system service routine definitions in the following section for the file system dependent and file system independent parameter areas.

Disk media and file system layout are described by the following structures. The data which are provided to the file system may depend on the level of file system support provided by the device driver attached to the block device. These structures are relevant only for local file systems.

```
/* file system independent - volume params */
struct vpfsi {
 unsigned long vpi_vid; /* 32 bit volume ID */
 unsigned long vpi_hDEV; /* handle to device driver */
 unsigned short vpi_bsize; /* sector size in bytes */
 unsigned long vpi_totsec; /* total number of sectors */
 unsigned short vpi_trksec; /* sectors / track */
 unsigned short vpi_nhead; /* number of heads */
 char vpi_text[12]; /* asciiz volume name */
}; /* vpfsi */
```

```
/* file system dependent - volume params */
struct vpfad {
 char vpd_work[36]; /* work area */
}; /* vpfad */
```

Per-disk current directories are described by the following structures. These structures can only be modified by the FSD during FS\_ATTACH and FS\_CHDIR operations.

```
/* file system independent - current dirs */
struct cdfsi {
 unsigned short cdi_hVPB; /* VPB handle for associated device */
 unsigned short cdi_end; /* offset to root of path */
 char cdi_flags; /* fs independent flags */
 char cdi_curdir[MAX]; /* text of current directory */
}; /* cdfsi */
```

```
/* file system dependent - current dirs */
struct cdfsd {
 char cdd_work[8]; /* work area */
}; /* cdfsd */
```

Open files are described by data initialized at file open time and discarded at the time of last close of all file handles which had been associated with that open instance of that file. There may be multiple open file references to the same file at any one time.

\* FSDs are required to support direct access opens. These are indicated by a bit set in the sfsi.sfi\_mode field.

```

/* file system independent - file instance */
struct sfsi {
 unsigned long sfi_mode; /* access/sharing mode */
 unsigned short sfi_hvpb; /* volume info. */
 unsigned short sfi_ctime; /* file creation time */
 unsigned short sfi_cdate; /* file creation date */
 unsigned short sfi_atime; /* file access time */
 unsigned short sfi_adata; /* file access date */
 unsigned short sfi_mtime; /* file modification time */
 unsigned short sfi_mdau; /* file modification date */
 unsigned long sfi_size; /* size of file */
 unsigned long sfi_position; /* read/write pointer */
 /* the following may be of use in sharing checks */
 unsigned short sfi_UID; /* user ID of initial opener */
 unsigned short sfi_PID; /* process ID of initial opener */
 unsigned short sfi_PDB; /* PDB (in 3.x box) of initial opener */
 unsigned short sfi_selfsfn; /* system file number of file instance */
 unsigned char sfi_tstamp; /* time stamps flags */
 unsigned short sfi_type; /* type of object opened */
}; /* sfsi */

/* file system dependent - file instance */
struct sfsd {
 char sfd_work[30]; /* work area */
}; /* sfsd */

```

The Program Data Block or program header, PDB (sfi pdb), is the unit of sharing for 3.x Box processes. For protect mode processes, the unit of sharing is the Process ID, PID (sfi pid). FSDs should use the combination <PDB, PID, UID> as indicating a distinct process.

The following data structures are the file system independent and dependent search records.

```

/* file system independent - file search parameters */
struct fsfi {
 unsigned short fsi_hvpb; /* volume info. */
}; /* fsfi */

/* file system dependent - file search parameters */
struct fsfd {
 char fsd_work[24]; /* work area */
}; /* fsfd */

```

Existing file systems that conform to the Standard Application Program Interface (Standard API) described in this section, may not necessarily support all the described information kept on a file basis. When such is the case, file system drivers are required to return to the application a null (zero) value for the unsupported parameter (when the unsupported data are a subset of the data returned by the API) or to return ENRONOT\_SUPPORTED (when all of the data returned by the API is unsupported).

#### 2.1.8.9.1 Time Stamping:

All time stamps on files are stamp and propagated to other SFT when the file is closed or committed (flushed). If a file is opened at time 1, written at time 2, and closed at time 3, the last write time will become time 3.

Subdirectories only have creation time stamps.

The sfi\_tstamp field contains six flags:

|           |     |    |                             |
|-----------|-----|----|-----------------------------|
| ST_SCREAT | EQU | 1  | ; stamp creation time       |
| ST_PCREAT | EQU | 2  | ; propagate creation time   |
| ST_SWRITE | EQU | 4  | ; stamp last write time     |
| ST_PWRITE | EQU | 8  | ; propagate last write time |
| ST_SREAD  | EQU | 16 | ; stamp last read time      |
| ST_PREAD  | EQU | 32 | ; propagate last read time  |

These flags are cleared when an SFT is created, and some of them may eventually be set by a file system worker. They are examined when the file is closed or flushed. For each time stamps, there are three meaningful actions:

| ST_Sxxx | ST_Pxxx | action                                      |
|---------|---------|---------------------------------------------|
| clear   | clear   | don't do anything                           |
| set     | set     | stamp and propagate (to other SFT and disk) |
| clear   | set     | don't stamp, but propagate existing value   |

#### 2.1.8.10 FSD calling conventions and requirements

Calling conventions between FS router, FSD, and FS helpers are:

- \* Arguments will be pushed in left-to-right order onto the stack.
- \* The callee is responsible for cleaning up the stack.
- \* Registers DS, SI, DI, BP, SS, SP are preserved.
- \* Return conditions appear in AX with the convention that AX == 0 indicates successful completion. AX != 0 indicates an error with the value of AX being the error code.

Interrupts must ALWAYS be enabled and the direction flag should be presumed to be undefined; calls to the FS helpers will change the direction flag at will.

In OS/2, file system drivers will always be called in kernel protect mode. This has the advantage of allowing the FSD to execute code without having to account for preemption; no preemption occurs when in kernel mode. While this greatly simplifies FSD structure, it forces the FSD to yield the CPU when executing long segments of code. In particular, an FSD must not hold the CPU for more than 2 milliseconds at a stretch. The FSD helper FSH\_YIELD is provided so that FSDs may relinquish the CPU.

The file system drivers cannot have any interrupt time activations. Since they occupy high, movable, and swappable memory, there is no guarantee upon addressability of the memory at interrupt time.

Each FS service routine may block.

#### \* 2.1.8.11 Error codes

- \* The FSD should use existing error codes when possible. New error codes must be in the range reserved for FSDs. The FS\_FSCTL interface must support returning information about new error codes.

The set of error codes for errors general to all FSDs is 0xEE00 - 0xEEFF. The following errors have been defined:

- \* ERROR\_VOLUME\_NOT\_MOUNTED = 0xEE00 - The FSD did not recognize the volume.

The set of error codes which are defined by each FSD are 0xEF00 - 0xFEFF.

#### 2.1.8.11.1 FS service routine command names:

The following table summarizes the commands associated with each FS entry point, and lists the names the FSD must use to export them. Note that names must be in all upper case as required by OS/2 naming convention.

| FS entry point     | OS/2 API supported           |
|--------------------|------------------------------|
| FS_ATTACH          | DOSQFSATTACH, DOSFSATTACH    |
| FS_CHDIR           | DOSCHDIR, DOSQCURDIR         |
| FS_CHGFILEPTR      | DOSCHGFILEPTR                |
| FS_CLOSE           | DOSCLOSE                     |
| FS_COMMIT          | DOSBUFRESET, DOSCLOSE        |
| FS_COPY            | DOSCOPY                      |
| FS_DELETE          | DOSDELETE                    |
| FS_EXIT            | DOSEXIT                      |
| FS_FILEATTRIBUTE   | DOSQFILEMODE, DOSSETFILEMODE |
| FS_FILEINFO        | DOSQFILEINFO, DOSSETFILEINFO |
| FS_FILEIO          | DOSFILEIO, DOSFILELOCKS      |
| FS_FINDCLOSE       | DOSFINDCLOSE                 |
| FS_FINDFIRST       | DOSFINDFIRST                 |
| FS_FINDFROMNAME    | --                           |
| FS_FINDNEXT        | DOSFINDNEXT                  |
| FS_FINDNOTIFYCLOSE | DOSFINDNOTIFYCLOSE           |
| FS_FINDNOTIFYFIRST | DOSFINDNOTIFYFIRST           |
| FS_FINDNOTIFYNEXT  | DOSFINDNOTIFYNEXT            |
| FS_FLUSHBUF        | DOSBUFRESET                  |
| FS_FSCTL           | DOSFSCTL                     |
| FS_FSINFO          | DOSQFSINFO, DOSSETFSINFO     |
| FS_INIT            | --                           |
| FS_IOCTL           | DOSDEVIOTL                   |
| FS_MKDIR           | DOSMKDIR                     |
| FS_MOVE            | DOSMOVE                      |
| FS_MOUNT           | --                           |
| FS_NEWSIZE         | DOSNEWSIZE                   |
| FS_NMPIPE          | --                           |
| FS_OPENCREATE      | DOSOPEN                      |
| FS_PATHINFO        | DOSQPATHINFO, DOSSETPATHINFO |
| FS_PROCESSNAME     | --                           |
| FS_READ            | DOSIREAD                     |
| FS_RMDIR           | DOSRMDIR                     |
| FS_SETSWAP         | --                           |
| FS_SHUTDOWN        | DOSSHUTDOWN                  |
| FS_WRITE           | DOSIWRITE                    |

The following OS/2 file system API routines are implemented entirely in OS/2 and do not need any services from the FSD: DOSDUPHANDLE, DOSQCURDISK, DOSQFHANDSTATE, DOSQHANDTYPE, DOSQVERIFY, DOSSCANENV, DOSSEARCHPATH, DOSSELECTDISK, DOSSETFHANDSTATE, DOSSETMAXFH, and DOSSETVERIFY.

#### 2.1.8.11.2 FS Entry Point Descriptions:

Each FS entry point has a distinct parameter list composed of those parameters needed by that particular entry. Parameters include:

- \* File pathnames
- \* Current disk/directory information
- \* Open file information
- \* Application data buffers
- \* Descriptions of file extended attributes
- \* Other parameters specific to an individual call

+ Most of the FS entry points have a level parameter for specifying the level of information they are provided or have to supply. FSDs must provide for additional levels which may be added in future versions of OS/2.

File system drivers which support hierarchical directory structures must use '\ ' and '/' as path name component separators. File system drivers which do not support hierarchical directory structures must reject as illegal any use of '\ ' or '/' in path names. The file names '.' and '..' are reserved for use in hierarchical directory structures for the current directory and the parent of the current directory respectively.

Unless otherwise specified in the descriptions below, data buffers may be accessed without concern for the accessibility of the data. In other words, OS/2 will either check buffers for accessibility and lock them, or transfer them into locally accessible data areas.

\* Simple parameters will be verified by the IFS router before the FS service routine is called.

### 2.1.8.11.3 FS\_ATTACH - Attach or Detach An FSD To A Drive or Device:

Purpose Attach or detach a remote drive or pseudo-device to an FSD.

#### Remote Drive:

```
int far pascal FS_ATTACH (flag, pDev, pvpfsd, pcdfsd, pParm, pLen)
unsigned short flag;
char far * pDev;
struct vpfsd far * pvpfsd;
struct cdfs far * pcdfsd;
char far * pParm;
unsigned short far * pLen;
```

#### Pseudo-device:

```
int far pascal FS_ATTACH (flag, pDev, pNull, pDevInfo, pParm, pLen)
unsigned short flag;
char far * pDev;
null ptr (OL) pNull;
unsigned long far * pDevInfo;
char far * pParm;
unsigned short far * pLen;
```

#### Where

flag indicates attach vs detach.

flag == 0 requests an attach. The FSD is being called to attach a specified drive or character device.

flag == 1 requests a detach.

flag == 2 requests the FSD to fill in the specified buffer with attachment information.

pDev pointer to the asciz text of either the drive (drive-letter followed by a colon) or to the character device (must be \DEV\device) that is being attached/detached/queried. The FSD does not need to verify this pointer.

pvpfsd/pNull pointer to structure of file-system dependent volume parameter information. When an attach/detach/query of a character device is requested, this pointer is null. When attaching a drive, this structure contains no data and is available for the FSD to store information needed to manage the remote drive. All subsequent FSD calls have access to the hVPB in one of the structures passed

in, so the FSD has access to this structure via the use of FSH\_GetVolParms. This structure will have its contents as the FSD had left them. When detaching or querying a drive, this structure contains the data as the FSD left them.

pcdfs/pDevInfo pointer to structure of file-system dependent working directory information for drives. When attaching a drive, this structure contains no data and is available for the FSD to store information needed to manage the working directory. All subsequent FSD calls (generated by API calls that reference this drive) are passed a pointer to this structure with contents left as the FSD left them. When detaching or querying a drive, this structure contains the data as the FSD left them. For character devices, pDevInfo points to a DWORD. When a device is attached, the DWORD contains no data, and can be used by the FSD to store a reference to identify the device later on during FS\_OpenCreate, when it is passed in to the FSD. When detaching or querying the device, this DWORD contains the data as the FSD left them.

pParm address of application parameter area. Addressing of this data area has not been validated by the kernel (See FSH\_PROBEBUF). When an attach is requested this will point to the API-specified user data block that contains information regarding the attach operation (e.g. passwords). For a query, the kernel will fill in part of the buffer, adjust the pointer, and call the FSD to fill in the rest. (See structure returned by DosQFsAttach, pParm will point to cbFSADData, the FSD should fill in cbFSADData and rgFSADData.) pParm must be verified, even in the query case.

pLen pointer to length of the application parameter area. For attach, this points to the length of the application data buffer. For query, this is the length of the remaining space in the application data buffer. Upon filling in the buffer, the FSD will set this to the length of the data returned. If the data returned is longer than the data buffer length, the FSD should set this value to be the length of the data that query could return. In this case, the FSD should also return ERROR\_BUFFER\_OVERFLOW. The FSD does not need to verify this pointer.

Remarks Local FSDs will never get called with attempts to attach or detach drives or queries about drives.

For remote FSDs called to do a detach, the kernel does not do any checking to see if there are any open references on the drive



(e.g. open or search references). It is entirely up to the FSD to decide whether it should allow the detach operation or not. Needless to say, this is not the normal thing to do.

#### 2.1.8.11.4 FS\_CHDIR - Change/Verify Directory Path:

**Purpose** Change or verify the directory path for the requesting process.

```
int far pascal FS_CHDIR (flag, pcdfsi, pcdfsd, pDir,
 iCurDirEnd)
unsigned short flag;
struct cdfs_i far * pcdfsi;
struct cdfs_d far * pcdfsd;
char far * pDir;
unsigned short iCurDirEnd;
```

**Where**

**flag** Indicates what action is to be taken on the directory.

flag == 0 indicates that an explicit directory-change request has been made.

flag == 1 indicates that the working directory need to be verified.

flag == 2 indicates that this reference to a directory is being freed.

\* The flag passed to the FSD will have a valid value.

\* **pcdfs\_i** pointer to file-system independent working directory structure. For flag == 0, this pointer points to the previous current directory on the drive. For flag == 1, this pointer points to the most-recent working directory on the drive. The cdi\_curdir field contains the text of the directory that is to be verified. For flag == 2, this pointer is null. The FSD MUST NEVER modify the cdfs\_i. The kernel handles all updates.

\* **pcdfs\_d** pointer to file-system dependent working directory structure. This is a place for the FSD to store information about the working directory. For flag == 0 or 1, this is the information left there by the FSD. The FSD is expected to update this information if the directory exists. For flag == 2, this is the information left there by the FSD.

\* **pDir** pointer to directory text. For flag == 0, this is the pointer to the directory. For flag == 1 or flag == 2, this pointer is null. The FSD does not need to verify this pointer.

\* **iCurDirEnd** index of the end of the current directory in pDir.

Microsoft Confidential  
OS/2 1.2 IFS Patent Documentation

This is used to optimize FSD path processing. If iCurDirEnd == -1 there is no current directory relevant to the directory text, i.e. a device. This parameter only has meaning for flag == 0.

Remarks The FSD should cache no information when the directory is the root. Root directories are a special case. They always exist, and never need validation. The kernel does not pass root directory requests to the FSD. And an FSD is not allowed to cache any information in the cdfs for a root directory. Under normal conditions, the kernel does not save the CDS for a root directory and builds one from scratch when it is needed. (One exception is where a validate cds fails, and the kernel sets it to the root, and zeroes out the cdfs. This CDS is saved and is cleaned up later.)

The following is information about the exact state of the cdfs and cdfs passed to the FSD for each flag value and guidelines about what an FSD should do upon receiving an FS\_CHDIR call.

```
* if (flag == 0) { /* Set new Current Directory */
*
* pcdfsi, pcdfsd = copy of CDS we're starting from; maybe useful
* as useful as starting point for verification.
*
* cdfs contents:
*
* hVPB - handle of Volume Parameter Block mapped to this
* drive
*
* end - end of 'root' portion of CurDir
*
* flags - various flags (indicating state of cdfs)
*
* IsValid - cdfs is unknown format (ignore contents)
*
* IsValid == 0x80
*
* IsRoot - cdfs is meaningless if CurDir = root
* (not kept)
*
* IsRoot == 0x40
*
* IsCurrent - cdfs is known format, but may not be
* current (medium may have been changed)
*
* IsCurrent == 0x20
*
* text - Current Directory Text
```

Microsoft Confidential  
OS/2 1.2 IFS Patent Documentation

icurdir = if Current Directory is in the path of the the new Current Directory, this is the index to the end of the Current Directory. If not, then this = -1 (Current Directory does not apply).

pDir = path to verify as legal directory.

DO THIS  
-----

Validate path named in pDir.  
/\* This means both that it exists AND that it's a dir.  
pcdfsi, pcdfsd, icurdir give old CDS, which may allow  
optimization \*/

```
if (Validate succeeds) {
 if (pDir != ROOT)
 store any cache information in area pointed to by
 pcdfsd.
 else
 do nothing!
 /* area pointed to by pcdfsd will be thrown away, so
 don't bother storing into it */
}
```

return success

```
else
 return failure
```

/\* Kernel will create new CDS using pDir data and pcdfsd data.

If the old CDS is valid, the kernel will take care of cleaning it up.

The FSD MUST NOT edit any structure other than the \*pcdfs area, with which it may do as it chooses. \*/

\*) /\* flag == 0 \*/

else if (flag == 1) { /\* Validate current CDS structure \*/

pcdfsi = pointer to copy of cdfs of interest.

pcdfs = pointer to copy of cdfs. Flags in cdfs indicate the state of this cdfs. It may be: (1) completely invalid (unknown format), (2) known format, but non-current information, (3) competely valid, or (4) all zero (root).

DO THIS  
-----

Validate that CDS still describes a legal directory (using

```

* cdi_text)
*
* if (valid) {
* update cdfsd if necessary
* return success
* /* kernel will copy cdfsd into real CDS */
* }
* else {
* if (cdi_isvalid)
* release any resources associated with cdfsd
* /* kernel will force Current Directory to root, and
* will zero out cdfsd in real CDS */
*
* return failure
* }
*
* /* The FSD MUST NOT modify any structure other than the cdfsd
* pointed to by pcdfsd. */
* }
*
* else if (flag == 2) { /* previous CDS no longer in use; being freed */
*
* pcdfsd = pointer to copy of cdfsd of CDS being freed.
*
* DO THIS
* -----
*
* Release any resources associated with the CDS.
* /* For example, if cdfsd (where pcdfsd points) contains a
* pointer to some FSD private structure associated with the
* CDS, that structure should be freed. */
*
* /* Kernel will not retain the cdfsd */
* }

```

#### 2.1.8.11.5 FS\_CHGFILEPTR - Move a file's position pointer:

Purpose Move a file's logical read/write position pointer.

```

| int far pascal FS_CHGFILEPTR (psffsi, psffsd, offset, type, IOflag)
| struct sffsi far * psffsi;
| struct sffsd far * psffsd;
| long offset;
| unsigned short type;
| unsigned short IOflag;

```

#### Where

```

* psffsi pointer to file-system independent portion of open file
* instance. The FSD uses the current file size or
* sfi_position along with offset and type to compute a new
* sfi_position. This is updated by the system.
*
* psffsd pointer to file-system dependent portion of open file
* instance. The FSD may store or adjust data as
* appropriate in this structure.
*
* offset signed offset to be added to the current file size or
* position to form the new position within the file.
*
* type indicates base of seek operation. type == 0 indicates
* seek relative to beginning of file. type == 1 indicates
* seek relative to current position within the file. type
* == 2 indicate seek relative to end of file. The value
* of type passed to the FSD will be valid.
*
| IOflag indicates information about the operation on the handle.
|
| IOflag == 0x0010 indicates write-through.
|
| IOflag == 0x0020 indicates no-cache.

```

\* Remarks The file system may want to take the seek operation as a hint that an I/O operation is about to take place at the new position and initiate a positioning operation on sequential access media or a read-ahead operation on other media.

\* Some 3xbox programs expect to be able to do a negative seek. OS/2 will pass these requests on to the FSD and will return an error for protect mode negative seek requests. Since a seek to a negative position is effectively a seek to a very large offset, it is suggested that the FSD return end-of-file for subsequent read requests.

FSDs must allow seeks to positions beyond end-of-file.

The information passed in IOflag is what was set for the handle during a DosOpen/DosOpen2 operation, or by a DosSetFHandState call.

#### 2.1.8.11.6 FS\_CLOSE - Close a File:

Purpose Closes the specified file handle.

```
int far pascal FS_CLOSE (type, IOflag, psffsi, psffsd)
unsigned short type;
unsigned short IOflag;
struct sffsi far * psffsi;
struct sffsd far * psffsd;
```

Where

type indicates what type of a close operation this is.

type == 0 indicates that this is not the final close of the file or device.

type == 1 indicates that is the final close of this file or device for this process.

type == 2 indicates that this is the final close for this file or device for the system.

IOflag indicates information about the operation on the handle.

IOflag == 0x0010 indicates write-through.

IOflag == 0x0020 indicates no-cache.

psffsi pointer to file-system independent portion of open file instance.

psffsd pointer to file-system dependent portion of open file instance.

Remarks Called on last close of a file.

Any reserved resources for this instance of the open file may be released. It may be assumed that all open files will be closed at process termination. That is not to say that this entry point will always be called at process termination for any files or devices open for the process.

A close operation should be interpreted by the FSD as meaning that the file should be committed to disk as appropriate.

Of the information passed in IOflag, the write-through bit is a

MANDATORY bit in that any data written to the block device must be put out on the medium before the device driver returns. The no-cache bit, on the other hand, is an advisory bit that says whether the data being transferred is worth caching or not.

#### 2.1.8.11.7 FS\_COMMIT - Commit a file's buffers to Disk:

**Purpose** Flush requesting process's cache buffers and update directory information for the file handle.

```
int far pascal FS_COMMIT (type, IOflag, psffsi, psffsd)
unsigned short type;
unsigned short IOflag;
struct sffsi far * psffsi;
struct sffsd far * psffsd;
```

**Where**

**type** indicates what type of a commit operation this is.

type == 1 indicates that this is a commit for a specific handle. This type is specified if FS\_COMMIT is called for a DosBufReset of specific handle.

type == 2 indicates that this is a commit due to a DosBufReset(-1).

**IOflag** indicates information about the operation on the handle.

IOflag == 0x0010 indicates write-through.

IOflag == 0x0020 indicates no-cache.

**psffsi** pointer to file-system independent portion of open file instance.

**psffsd** pointer to file-system dependent portion of open file instance.

**Remarks** Only called via DosBufReset. OS/2 reserves the right to call FS\_COMMIT even if no changes have been made to the file.

For DosBufReset (-1), FS\_COMMIT will be called for each handle the calling process has open on the FSD.

The FSD should update access and modification times, if appropriate.

Any locally cached information about the file must be output to the media. The directory entry for the file is to be updated from the sffsi and sffsd structures.

Since MiniFSDs used to boot IFSs are read-only file systems, they need not support the FS\_COMMIT call.

Of the information passed in IOflag, the write-through bit is a MANDATORY bit in that any data written to the block device must be put out on the medium before the device driver returns. The no-cache bit, on the other hand, is an advisory bit that says whether the data being transferred is worth caching or not.

The FSD should copy all supported time stamps from the SFT to the disk. Beware that the last read time stamp may need to be written to the disk even though the file is clean. After this is done, the FSD should clear the sft.timestamp field to avoid having to write to the disk again if the user calls commit repeatedly without changing any of the time stamps.

If the disk is not writeable and only the last read time stamp has changed, the FSD should either issue a warning or ignore the error. This relieves the user from having to un-protect an FSD floppy disk in order to read the files on it.

#### 2.1.8.11.8 FS\_COPY - Copy a file:

Purpose Copy specified file or subdirectory to specified target.

```
int far pascal FS_COPY(flag,pcdfsi,pcdfsd,
 pSrc, iSrcCurDirEnd,
 pDst, iDstCurDirEnd)

unsigned short flag;
struct cdfsi far * pcdfsi;
struct cdfsd far * pcdfsd;
char far * pSrc;
unsigned short iSrcCurDirEnd;
char far * pDst;
unsigned short iDstCurDirEnd;
```

Where

flag bitmask controlling copy

0x0001 specifies that an existing target file/directory should be replaced (see explanation above under DosCopy).

0x0002 specifies that a source file will be appended to the destination file (see explanation above under DosCopy).

All other bits reserved.

pcdfsi Pointer to the file-system independent working directory structure.

pcdfsd Pointer to the file-system dependent working directory structure.

pSrc Pointer to ASCIIZ name of source file/directory.

iSrcCurDirEnd index of the end of the current directory in pSrc. If = -1, there is no current directory relevant to the source name.

pDst Pointer to ASCIIZ name of destination file/directory.

iDstCurDirEnd index of the end of the current directory in pDst. If = -1, there is no current directory relevant to the destination name.

Remarks The file specified in sourcename should be copied to the targetfile if possible.

There will be no assurances that the files specified are not currently open. File system drivers will have to assure consistency of file allocation information and directory entries.

The file system driver should return the special error ERROR\_CANNOT\_COPY if it cannot perform the copy because:

- \* it doesn't know how
- \* the source and target are on different volumes
- \* of any other reason for which it would make sense for its caller to perform the copy operation manually.

Returning ERROR\_CANNOT\_COPY indicates to its caller that it should attempt to perform the copy operation manually; any other error will be returned directly to the caller of DOSCOPY.

FS\_COPY needs to check that certain types of illegal copying operations are not performed. A directory cannot be copied to itself or to one of its subdirectories. This is especially critical in situations where two different fully-qualified pathnames can refer to the same file or directory. (For example, if X: is redirected to \\SERVER\SHARE, then X:\PATH and \\SERVER\SHARE\PATH refer to the same object.)

The behavior of FS\_COPY should match the behavior of the generic DosCopy routine.

See errors under DOSCOPY for other error codes that can be returned.

#### 2.1.8.11.9 FS\_DELETE - Delete a File:

**Purpose** Removes a directory entry associated with a filename.

```
int far pascal FS_DELETE (pcdfsi, pcdfsd, pFile, iCurDirEnd)
struct cdfsi far * pcdfsi;
struct cdfsd far * pcdfsd;
char far * pFile;
unsigned short iCurDirEnd;
```

**Where**

pcdfsi pointer to file-system independent working directory structure.

pcdfsd pointer to file-system dependent working directory structure.

pFile pointer to asciiz name of file/directory. The FSD does not need to validate this pointer.

iCurDirEnd index of the end of the current directory in pFile. This is used to optimize FSD path processing. If iCurDirEnd == -1 there is no current directory relevant to the name text, i.e. a device.

**Remarks** The file(s) specified in filename should be deleted.

The deletion of a file opened in compatibility mode in the 3xbox by the same process requesting the delete is supported. OS/2 will call FS\_CLOSE for the file before issuing the call to FS\_DELETE.

The filename may not contain wildcard characters.

#### 2.1.8.11.10 FS\_EXIT - End of process:

**Purpose** Release FSD resources still held after process termination.

```
void far pascal FS_EXIT (uid, pid, pdb);
unsigned short uid;
unsigned short pid;
unsigned short pdb;
```

#### Where

- \* uid user id of process. This will be a valid value.
- \* pid process id of process. This will be a valid value.
- \* pdb 3x box process id of process. This will be a valid value.
- \*

**Remarks** This call is not needed to release file resources since all files are closed on process termination, but it may be helpful if resources are being held due to unterminated searches (in the case of searches initiated from the 3.x box).

#### 2.1.8.11.11 FS\_FILEATTRIBUTE - Query/Set File Attribute:

**Purpose** Query/set the attribute of the specified file.

```
int far pascal FS_FILEATTRIBUTE (flag, pcdfsi, pcdfsd,
 pname, iCurDirEnd, pAttr)
unsigned short flag;
struct cdfs_i far * pcdfsi;
struct cdfs_d far * pcdfsd;
char far * pname;
unsigned short iCurDirEnd;
unsigned short far * pAttr;
```

**flag** indicates retrieval of attributes vs setting attributes

flag == 0 indicates retrieving the attribute.

flag == 1 indicates setting the attribute.

All other values reserved.

The value of flag passed to the FSD will be valid.

**pcdfs\_i** pointer to file-system independent working directory structure.

**pcdfs\_d** pointer to file-system dependent working directory structure.

\* **pName** pointer to asciiz name of file/directory. The FSD does  
\* not need to validate this pointer.

\* **iCurDirEnd** index of the end of the current directory in pName.  
\* This is used to optimize FSD path processing. If  
\* iCurDirEnd == -1 there is no current directory relevant  
\* to the name text, i.e. a device.

\* **pAttr** pointer to attribute. For flag == 0, the FSD should  
\* store the attribute in the indicated location. For flag  
\* == 1, the FSD should retrieve the attribute from this  
\* location and set it in the file/directory. The FSD does  
\* not need to validate this pointer.



#### 2.1.8.11.12 FS\_FILEINFO : Query/Set a File's Information:

**Purpose** Returns information for a specific file.

```
int far pascal FS_FILEINFO (flag, psffsi, psffsd,
 level, pData, cbData, IOflag)
unsigned short flag;
struct sffsi far * psffsi;
struct sffsd far * psffsd;
unsigned short level;
char far * pData;
unsigned short cbData;
unsigned short IOflag;
```

**Where**

flag indicates retrieval of information vs setting information.

flag == 0 indicates retrieving information.

flag == 1 indicates setting information.

All other values reserved.

The value of flag passed to the FSD will be valid.

psffsi pointer to file-system independent portion of open file instance.

psffsd pointer to file-system dependent portion of open file instance.

level information level to be returned. Level selects among a series of structures of data to be returned.

pData address of application data area. Addressing of this data area has not been validated by the kernel (See FSH\_PROBEBUF). When retrieval (flag == 0) is specified, the FSD will place the information into the buffer. When outputting information to a file (flag == 1), the FSD will retrieve that data from the application buffer.

cbData length of the application data area. For flag == 0, this is the length of the data the application wishes to retrieve. If there is not enough room for the entire level of data to be returned, the FSD will return ERROR\_BUFFER\_OVERFLOW. For flag == 1, this is the length of data to be applied to the file.

IOflag indicates information about the operation on the handle.

IOflag == 0x0010 indicates write-through.

IOflag == 0x0020 indicates no-cache.

**Remarks** If setting time on file, copy new time/date into SFT, then set ST\_PCREAT, ST\_PWRITE, and ST\_PREAD but clear ST\_SCREAT, ST\_SWRITE, and ST\_SREAD. If querying time on file, simply copy time stamps from directory entry into SFT.

Of the information passed in IOflag, the write-through bit is a MANDATORY bit in that any data written to the block device must be put out on the medium before the device driver returns. The no-cache bit, on the other hand, is an advisory bit that says whether the data being transferred is worth caching or not. The IOflag bit applies to both the data in the file and to the EAs attached to the file as well.

The supported information levels are described in the OS/2 1.2 API description.

#### 2.1.8.11.13 FS\_FILEIO - Multi-function File I/O:

Purpose Perform multiple lock, unlock, seek, read, and write I/O.

```
int far pascal FS_FILEIO (psffsi, psffsd,
 pCmdList, cbCmdList, poError, IOflag)
*
* struct sffsi far * psffsi;
* struct sffsd far * psffsd;
* char far * pCmdList;
* unsigned short cbCmdList;
* unsigned short far * poError;
* unsigned short IOflag;
```

Where

psffsi pointer to file-system independent portion of open file instance.

psffsd pointer to file-system dependent portion of open file instance.

pCmdList pointer to command list that contains entries indicating what commands will be performed. Each individual operation (CmdLock, CmdUnlock, CmdSeek, CmdIO) is performed as atomic operations until all are complete or until one fails. CmdLock executes a multiple range lock as an atomic operation. Unlike CmdLock, CmdUnlock cannot fail as long as the parameters to it are correct, and the calling application had done a Lock earlier, and so it can be viewed as atomic. The validity of the user address has not been verified (see FSH\_PROBEBUF).

For CmdLock, the command format is:

```
CmdLock Struc
Cmd dw 0 ; 0 for lock operations
LockCnt dw ? ; Number of locks that follow
Timeout dd ? ; ms timeout for lock success
CmdLock Ends
```

which is followed by a series of records of the following format:

```
Lock Struc
Share dw ? ; 0 for exclusive, 1 for read-only access
Start dd ? ; start of lock region
Length dd ? ; length of lock region
Lock Ends
```

If a lock within a CmdLock causes a timeout, none of the other locks within the scope of CmdLock are in force since the lock operation is viewed as atomic.

CmdLock.Timeout is the count in milliseconds, until the requesting process is to resume execution if the requested locks are not available. If CmdLock.Timeout == 0, there will be no wait. If CmdLock.Timeout < 0xFFFFFFFF it is the number of milliseconds to wait until the requested locks become available. If CmdLock.Timeout == 0xFFFFFFFF then the thread will wait indefinitely until the requested locks become available.

Lock.Share defines the type of access other processes may have to the file-range being locked. value == 0, other processes have 'No-Access' to the locked range. value == 1, other processes have 'Read-Only' access to the locked range.

For CmdUnlock, the command format is:

```
CmdUnlock Struc
Cmd dw 1 ; 1 for unlock operations.
UnlockCnt dw ? ; Number of unlocks that follow
CmdUnlock Ends
```

which is followed by a series of records of the following format:

```
Unlock Struc
Start dd ? ; start of locked region
Length dd ? ; length of locked region
Unlock Ends
```

For CmdSeek, the command format is:

```
CmdSeek Struc
Cmd dw ? ; 2 for seek operation
Method dw ? ; 0 for absolute,
; 1 for relative to current,
; 2 for relative to EOF.
Position dd ? ; file seek position or delta
Actual dd ? ; actual position seeked to
CmdSeek Ends
```

For CmdIO, the command format is:

```
CmdIO Struc
Cmd dw ? ; 3 for read, 4 for write
Buffer# dd ? ; ptr to the data buffer
BufferLen dw ? ; number of bytes requested
Actual dw ? ; number of bytes actually
```

CmdIO      Ends      ; transferred

```
* cbCmdList length in bytes of the command list.
*
* poError offset within the command list of the command that
+ caused the error. This offset is relative to the
+ beginning of the command list. This field only has value
* when an error occurs. The validity of the user address
| has not been verified (see FSH_PROBEBUF).
|
| IOflag indicates information about the operation on the handle.
|
| IOflag == 0x0010 indicates write-through.
|
| IOflag == 0x0020 indicates no-cache.
|
Remarks This function provides a simple mechanism for combining the
 following operations into a single request and providing improved
 performance particularly in a networking environment.
|
| File Systems that do not have the FileIO bit in their FS_ATTRIBUTE
| point will never see this call. The command list will be parsed
| by the IFS router; the FSD will see only FS_CHGFILEPTR, FS_READ,
| FS_WRITE calls.
|
| File systems that have the FileIO bit in their attribute field
| will see this call in its entirety. The atomicity guarantee
| applies only to the commands themselves and not to the list as a
| whole.
|
| Of the information passed in IOflag, the write-through bit is a
| MANDATORY bit in that any data written to the block device must be
| put out on the medium before the device driver returns. The
| no-cache bit, on the other hand, is an advisory bit that says
| whether the data being transferred is worth caching or not.
```

#### 2.1.8.11.14 FS\_FINDCLOSE - Directory Read (Search) Close:

Purpose      Provides the mechanism for an FSD to release resources allocated  
            on behalf of FS\_FINDFIRST and FS\_FindNext.

```
* int far pascal FS_FINDCLOSE (pfsfsi, pfsfsd)
* struct fsfsi far * pfsfsi;
* struct fsfsd far * pfsfsd;
```

Where

pfsfsi      pointer to file-system independent file search  
            structure. The FSD should not update this structure.

pfsfsd      pointer to file-system independent file search  
            structure. The FSD may use this to store information  
            about continuation of the search.

Remarks      DOSFINDCLOSE has been called on the handle associated with the  
            search buffer. Any file system related information may be  
            released.

If FS\_FINDFIRST for a particular search returns an error, an  
FS\_FINDCLOSE for that search will not be issued.

# 2.1.8.11.15 FS\_FINDFIRST, - Find First Matching File Name:

\* Purpose Find first occurrence of a file name in a directory.

```

* int far pascal FS_FINDFIRST (pcdfsi, pcdfsd, pName, iCurDirEnd,
* attr, pfsfsi, pfsfsd,
* pData, cbData, pcMatch,
* level, flags)
*
* struct cdfsi far * pcdfsi;
* struct cdfsd far * pcdfsd;
* char far * pName;
* unsigned short iCurDirEnd;
* unsigned short attr;
* struct fsfsi far * pfsfsi;
* struct fsfsd far * pfsfsd;
* char far * pData;
* unsigned short cbData;
* unsigned short far * pcMatch;
* unsigned short level;
* unsigned short flags;

```

Where

pcdfsi pointer to file-system independent working directory structure.

pcdfsd pointer to file-system dependent working directory structure.

pName pointer to asciz name of file/directory. Wildcard characters are allowed only in the last component. The FSD does not need to validate this pointer.

iCurDirEnd index of the end of the current directory in pName. This is used to optimize FSD path processing. If iCurDirEnd == -1 there is no current directory relevant to the name text, i.e. a device.

attr bit field that governs the match. Any directory entry whose attribute bit mask is a subset of attr and whose name matches that in pName should be returned. For example, an attribute of system and hidden is passed in. A file with the same name and an attribute of system is found. This file should be returned. A file with the same name and no attributes (a regular file) would also be returned. The attributes read-only and file archive will not be passed in and should be ignored when comparing directory attributes. The value of attr passed to the FSD will be valid.

pfsfsi pointer to file-system independent file search structure. The FSD should not update this structure.

pfsfsd pointer to file-system independent file search structure. The FSD may use this to store information about continuation of the search.

pData address of application data area. Addressing of this data area has not been validated by the kernel (See FSH\_PROBEBUF). The FSD will fill in this area with a set of packed, variable-length structures that contain the requested data and matching file names.

cbData length of the application data area in bytes.

pcMatch pointer to number of matching entries. The FSD will return at most this number of entries. The FSD will store into it the number of entries actually placed in the data area. The FSD does not need to validate this pointer.

level information level to be returned. Level selects among a series of structures of data to be returned. The level passed to the FSD will be valid.

flags indicates whether to return file position information.

flags == 0 indicates that file position information should not be returned and the information format described under DosFindFirst should be used.

flags == 1 indicates that file position information should be returned and the information format described below should be used. the flag passed to the FSD will have a valid value.

\* Remarks

For flags == 1, the FSD must store in the first dword of the per-file attributes structure adequate information to allow the search to be resumed from the file by calling FS\_FINDFROMNAME. For example, an ordinal representing the file's position in the directory could be stored. If the filename must be used to restart the search, the dword may be left blank.

For level 0x0001 and flags == 0, directory information for FS\_FINDFIRST (find record) is returned in the following format:

```
struct {
 unsigned short dataCreate;
 unsigned short timeCreate;
 unsigned short dateAccess;
 unsigned short timeAccess;
 unsigned short dateWrite;
 unsigned short timeWrite;
 long cbEOF;
 long cbAlloc;
 unsigned short attr;
 unsigned char cbName;
 unsigned char szName[];
};
```

For level 0x0001 and flags == 1, directory information for FS\_FINDFIRST (find record) is returned in the following format:

```
struct {
 long position;
 unsigned short dataCreate;
 unsigned short timeCreate;
 unsigned short dateAccess;
 unsigned short timeAccess;
 unsigned short dateWrite;
 unsigned short timeWrite;
 long cbEOF;
 long cbAlloc;
 unsigned short attr;
 unsigned char cbName;
 unsigned char szName[];
};
```

The other information levels have similar format, with the position the first field in the structure for flags == 1.

If FS\_FINDFIRST for a particular search returns an error, an FS\_FINDCLOSE for that search will not be issued.

Sufficient information to find the next matching directory entry must be saved in the fsfsd structure.

In the case where directory entry information overflows the pData area, the FSD should be able to continue the search from the entry which caused the overflow on the next FS\_FINDNEXT or FS\_FINDFROMNAME..

In the case of a global search in a directory, the first two entries in that directory as reported by FSD should be '.' and

+'..' (current and parent directories).

2.1.8.11.16 FS\_FINDFROMNAME - Find Matching File Name Starting from Name:

Purpose Find occurrence of a file name in a directory beginning from position or pName.

```
int far pascal FS_FINDFROMNAME (pfsfsi, pfsfsd, pData, cbData,
 pcMatch, position, pName)
```

```
struct fsfsi far * pfsfsi;
struct fsfsd far * pfsfsd;
char far * pData;
unsigned short cbData;
unsigned short far * pcMatch;
unsigned short level;
unsigned long position;
char far * pName;
unsigned short flags;
```

Where

pfsfsi pointer to file-system independent file search structure. The FSD should not update this structure.

pf:fsd pointer to file-system dependent file search structure. The FSD may use this to store information about continuation of the search.

pData address of application data area. Addressing of this data area has not been validated by the kernel (See FSH\_PROBEBUF). The FSD will fill in this area with a set of packed, variable-length structures that contain the requested data and matching file names in the format required for DosFindFirst/Next.

cbData length of the application data area in bytes.

pcMatch pointer to number of matching entries. The FSD will return at most this number of entries. The FSD will store into it the number of entries actually placed in the data area. The FSD does not need to validate this pointer.

level information level to be returned. Level selects among a series of structures of data to be returned. The level passed to the FSD will be valid.

position the file-system specific information about where to restart the search from. This information was returned by the FSD in the ResultBuf for a DosFindFirst2/Next/FromName call.

pName is the filename to from which to continue the search.

The FSD does not need to validate this pointer.

flags indicates whether to return file position information. The flag passed to the FSD will have a valid value.

+ Remarks The FSD may use either the position or filename to determine the position from which to resume the directory search. So the FSD need not return position if it uses name and vice versa.

For flags == 1, the FSD must store in the position field adequate information to allow the search to be resumed from the file by calling FS\_FINDFROMNAME. See FS\_FINDFIRST for a description of the data format.

The FSD must ensure that enough information is stored in the fsfsd structure to enable it to continue the search.

2.1.8.11.17 FS\_FINDNEXT - Find Next Matching File Name:

\* Purpose Find next occurrence of a file name in a directory.

```
* int far pascal FS_FINDNEXT (pfsfsi, pfsfsd, pData, cbData, pcMatch,
| level, flags)
*
* struct fsfsi far * pfsfsi;
* struct fsfsd far * pfsfsd;
* char far * pData;
* unsigned short cbData;
* unsigned short far * pcMatch;
| unsigned short level;
| unsigned short flags;
```

Where

```
| pfsfsi pointer to file-system independent file search
| structure. The FSD should not update this structure.
|
| pfsfsd pointer to file-system dependent file search structure.
| The FSD may use this to store information about
| continuation of the search.
|
| pData address of application data area. Addressing of this
| data area has not been validated by the kernel (See
| FSH_PROBEBUF). The FSD will fill in this area with a
| set of packed, variable-length structures that contain
| the requested data and matching file names.
|
| cbData length of the application data area in bytes.
|
| pcMatch pointer to number of matching entries. The FSD will
| return at most this number of entries. The FSD will
| store into it the number of entries actually placed in
| the data area. The FSD does not need to validate this
| pointer.
|
| level information level to be returned. Level selects among a
| series of structures of data to be returned. The level
| passed to the FSD will be valid.
|
| flags indicates whether to return file position information.
```

```
| Remarks For flags == 1, the FSD must store in the position field adequate
| information to allow the search to be resumed from the file by
| calling FS_FINDFROMNAME. See FS_FINDFIRST for a description of
| the data format.
```

```
| The level passed to FS_FINDNEXT will be the same level as that
| passed to FS_FINDFIRST to initiate the search.
```

```
* Sufficient information to find the next matching directory entry
* must be saved in the fsfsd structure.
```

```
+ The FSD should take care of the case where the pData are overflow
+ may occur. FSD should be able to start the search from the same
+ entry for the next FS_FINDNEXT as the one for which the overflow
+ occurred.
```

```
+ In the case of a global search in a directory, the first two
+ entries in that directory as reported by FSD should be '.' and
+ '..' (the current and the parent directories).
```

#### 2.1.8.11.18 FS\_FINDNOTIFYCLOSE - Close Find-Notify Handle:

**Purpose** Closes the association between a 'Find-Notify' handle and a DOSFINDNOTIFYFIRST or DOSFINDNOTIFYNEXT function. Provides the mechanism for an FSD to release resources allocated on behalf of FS\_FINDNOTIFYFIRST and FS\_FINDNOTIFYNEXT.

```
*
* int far pascal FS_FINDNOTIFYCLOSE (handle)
* unsigned short handle;
```

**Where**

```
* handle directory handle. This handle was returned by the FSD
* and is associated with a previous FS_FINDNOTIFYFIRST or
* FS_FINDNOTIFYNEXT call.
```

**Remarks** FS\_FINDNOTIFYCLOSE has been called on the handle associated with a FS\_FINDNOTIFYFIRST. Any file system related information may be released.

#### 2.1.8.11.19 FS\_FINDNOTIFYFIRST - Start monitoring directory for changes:

**Purpose** Start monitoring a directory for changes.

```
int far pascal FS_FINDNOTIFYFIRST (pcdfsi, pcdfsd, pName, iCurDirEnd,
 attr, pHandle,
 pData, cbData, pcMatch,
 level, timeout)
*
*
* struct cdfsi far * pcdfsi;
* struct cdfsd far * pcdfsd;
* char far * pName;
* unsigned short iCurDirEnd;
* unsigned short attr;
* unsigned short far * pHandle;
* char far * pData;
* unsigned short cbData;
* unsigned short far * pcMatch;
* unsigned short level;
* unsigned long timeout;
```

**Where**

pcdfsi pointer to file-system independent working directory structure.

pcdfsd pointer to file-system dependent working directory structure.

pName pointer to asciiz name of file/directory. Wildcard characters are allowed only in the last component. The FSD does not need to verify this pointer.

iCurDirEnd index of the end of the current directory in pName. This is used to optimize FSD path processing. If iCurDirEnd == -1 there is no current directory relevant to the name text, i.e. a device.

attr bit field that governs the match. Any directory entry whose attribute bit mask is a subset of attr and whose name matches that in pName should be returned. See FS\_FINDFIRST for explanation.

pHandle pointer to directory handle. The FSD must allocate a handle for the directory monitoring continuation information and store it here. This handle will be passed to FS\_FINDNOTIFYNEXT to continue directory monitoring. The FSD does not need to verify this pointer.

pData address of application data area. Addressing of this



Microsoft Confidential  
OS/2 1.2 IFS Patent Documentation

data area has not been validated by the kernel (See FSH\_PROBEBUF). The FSD will fill in this area with a set of packed, variable-length structures that contain the requested data and matching file names.

cbData    length of the application data area in bytes.

pcMatch   pointer to number of matching entries. The FSD will return at most this number of entries. The FSD will store into it the number of entries actually placed in the data area. The FSD does not need to verify this pointer.

level    information level to be returned. Level selects among a series of structures of data to be returned. See DOSFINDNOTIFYFIRST for information. The level passed to the FSD will be valid.

timeout   millisecond timeout. The FSD will wait until either the timeout has expired, the buffer is full, or the specified number of entries returned before returning to the caller.

Microsoft Confidential  
OS/2 1.2 IFS Patent Documentation

2.1.8.11.20 FS\_FINDNOTIFYNEXT - Resume reporting directory changes:

Purpose    Resume reporting of directory or file changes.

```
int far pascal FS_FINDNOTIFYNEXT (handle, pData, cbData, pcMatch,
 level, timeout)
unsigned short handle;
char far * pData;
unsigned short cbData;
unsigned short far * pcMatch;
unsigned short level;
unsigned long timeout;
```

Where

handle    directory handle. This handle was returned by the FSD and is associated with a previous FS\_FINDNOTIFYFIRST or FS\_FINDNOTIFYNEXT call.

pData    address of application data area. Addressing of this data area has not been validated by the kernel (See FSH\_PROBEBUF). The FSD will fill in this area with a set of packed, variable-length structures that contain the requested data and matching file names.

cbData    length of the application data area in bytes.

pcMatch   pointer to number of matching entries. The FSD will return at most this number of entries. The FSD will store into it the number of entries actually placed in the data area. The FSD does not need to verify this pointer.

level    information level to be returned. Level selects among a series of structures of data to be returned. See DOSFINDNOTIFYFIRST for information. The level passed to the FSD will be valid.

timeout   millisecond timeout. The FSD will wait until either the timeout has expired, the buffer is full, or the specified number of entries returned before returning to the caller.

Remarks   pcMatch is the number of changes required to directories or files that match the pName target and attr specified during a related, previous FS\_FINDNOTIFYFIRST. The file system uses this field to return the number of changes that actually occurred since the issue of the present FS\_FINDNOTIFYNEXT.

The level passed to FS\_FINDNOTIFYNEXT will be the same level as that passed to FS\_FINDNOTIFYFIRST to initiate the search.

#### 2.1.8.11.21 FS\_FLUSHBUF - Commit file buffers:

**Purpose** Flushes cache buffers for a specific volume.

```
int far pascal FS_FLUSHBUF (hVPB, flag)
unsigned short hVPB;
unsigned short flag;
```

**Where**

hVPB handle to volume for flush.

flag indicator for discarding of cached data.

flag == 0 indicates, cached data may be retained.

flag == 1 indicates the FSD will discard any cached data after flushing it to the specified volume.

All other values reserved.

#### 2.1.8.11.22 FS\_FCTL - File System Control:

**Purpose** Allow an extended standard interface between an application and a file system driver.

```
int far pascal FS_FCTL (pArgdat, iArgType, func,
 pParm, lenParm, plenParmIO,
 pData, lenData, plenDataIO)
```

```
union argdat far * pArgDat
{
 unsigned short iArgType;
 unsigned short func;
 char far * pParm;
 unsigned short lenParm;
 unsigned short far * plenParmIO;
 char far * pData;
 unsigned short lenData;
 unsigned short far * plenDataIO;
}
```

**Where**

pArgDat pointer to union whose contents depend on iArgType. Union is:

```
union argdat {
 /* pArgType = 1, FileHandle directed case */
 struct sf {
 struct sffsi far * psfsi;
 struct sffsd far * psfsd;
 };

 /* pArgType = 2, Pathname directed case */
 struct cd {
 struct cdfsi far * pcdfsi;
 struct cdfsd far * pcdfsd;
 char far * pPath;
 unsigned short iCurDirEnd;
 };

 /* pArgType = 3, FSD Name directed case */
 /* garbage */
};
```

iArgType indicator of argument type.

\* iArgType = 1 means that pArgDat->sf.psfsi and pArgDat->sf.psfsd point to an sffsi and sffsd, respectively.

\* iArgType = 2 means that pArgDat->cd.pcdfsi and pArgDat->cd.pcdfsd point to a cdfsi and cdfsd, pArgDat->cd.pPath points to a canonical pathname, and pArgDat->cd.iCurDirEnd gives the index of the end of the current directory in pPath. The FSD does not need to verify the pPath pointer.

\* iArgType = 3 means that the call was FSD name routed, and pArgDat is a NULL pointer.

func indicator of function to perform.

func == 1 indicates request for new error code information.

func == 2 indicates request for maximum EA size and EA list size supported. This will be returned in the buffer pointed to by pData in the following format:

```
EASizeBufStruc {
 unsigned short easb_MaxEASize /* Max. size EA supported */
 unsigned long easb_MaxEAListSize /* Max. full supported */
}
```

pParm address of application input parameter area. Addressing of this data area has not been validated by the kernel (See FSH\_PROBEBUF).

lenParm byte length of application input parameter area.

plenParmIO On input, contains the length in bytes of the parameters being passed to the FSD in pParm. On return, contains the length in bytes of data returned in pParm by the FSD. The length of the data returned by the FSD in pParm must not exceed the length in lenParm. Addressing of this area have not been validated by the kernel (see FSH\_PROBEBUF).

pData address of application output data area. Addressing of this data area has not been validated by the kernel (See FSH\_PROBEBUF).

lenData byte length of application output data area.

plenDataIO On input, contains the length in bytes of the data being passed to the FSD in pData. On return, contains the length in bytes of data returned in pData by the

FSD. The length of the data returned by the FSD in pData must not exceed the length in lenData. Addressing of this area has not been validated by the kernel (see FSH\_PROBEBUF).

Remarks The accessibility of the parameter and data buffers and the locations of the length of the data and parameter areas has not been validated by the kernel. FS\_PROBEBUF must be used.

All FSDs must support func == 1 to return new error code information, and func == 2 to return the limits of the Extended Attribute sizes.

For func == 1, the error code is passed to the FSD in the first word of the parameter area. On return, the first word of the data area contains the length of the asciiz string containing an explanation of the error code. The data area contains the asciiz string beginning at the second word. All FSDs are required to support func == 1.

#### 2.1.8.11.23 FS\_FSINFO - File System Information:

Purpose Returns/Sets information for a filesystem device.

```
int far pascal FS_FSINFO (flag, hVPB, pData, cbData, level)
unsigned short flag;
unsigned short hVPB;
char far * pData;
unsigned short cbData;
unsigned short level;
```

Where

flag indicates retrieval of information vs setting information.

flag == 0 indicates retrieving information.

flag == 1 indicates setting information on the media.

All other values reserved.

hVPB handle to volume of interest.

pData address of application output data area. Addressing of this data area has not been validated by the kernel (See FSH\_PROBEBUF).

cbData length of the application data area. For flag == 0, this is the length of the data the application wishes to retrieve. If there is not enough room for the entire level of data to be returned, the FSD will return ERROR\_BUFFER\_OVERFLOW. For flag == 1, this is the length of data to be sent to the file system.

level information level to be returned. Level selects among a series of structures of data to be returned or set. See DOSQFSINFO and DOSSETFSINFO for information.

Remarks None.

#### 2.1.8.11.24 FS\_INIT - File system driver initialization:

**Purpose** Request file system driver initialization.

```
int far pascal FS_INIT (szParm, DevHelp, pMiniFSD)
char far * szParm;
unsigned long DevHelp;
unsigned long far * pMiniFSD;
```

**Where**

```
szParm pointer to asciz parameters following the config.sys
IFS= command that loaded the FSD. The FSD does not need
to verify this pointer.

DevHelp address of the kernel entry point for the DevHelp
routines. This is used exactly as the device driver
DevHelp address, and can be used by an FSD that needs
access to some of the device helper services.

pMiniFSD pointer to data passed between the mini-FSD and the FSD
or null.
```

**Remarks** This call is made during system initialization to allow the FSD to perform actions necessary for beginning operation. The FSD may successfully initialize by returning 0 or may reject installation (invalid parameters, incompatible hardware, etc) by returning the appropriate error code. If rejection is selected, all FSD selectors and segments are released.

```
pMiniFSD will be null except when booting from a volume managed by
an FSD and the exported name of the FSD matches the exported name
of the mini-FSD. In this case, pMiniFSD will point to data
established by the mini-FSD (see mFS_INIT).
```

#### 2.1.8.11.25 FS\_IOCTL - I/O Control for Devices:

**Purpose** Perform control functions on the device specified by the opened device handle.

```
int far pascal FS_IOCTL (psffsi, psffsd, cat, func,
pParm, lenParm,
pData, lenData)

struct sffsi far * psffsi;
struct sffsd far * psffsd;
unsigned short cat;
unsigned short func;
char far * pParm;
unsigned short lenParm;
char far * pData;
unsigned short lenData;
```

**Where**

```
psffsi pointer to file-system independent portion of open file
instance.

psffsd pointer to file-system dependent portion of open file
instance.

cat category of function to be performed.

func function within category to be performed.

pParm address of application input parameter area. Addressing
of this data area has not been validated by the kernel
(See FSH_PROBEBUF). A null value indicates that the
parameter is unspecified for this function.

lenParm byte length of application input parameter area. If
lenParm is 0 and pParm is not null, it means the data
buffer length is unknown due to the request being
submitted via an old IOCTL or DOSDEVIOTL interface.

pData address of application output data area. Addressing
of this data area has not been validated by the kernel
(See FSH_PROBEBUF). A null value indicates that the
parameter is unspecified for this function.

lenData byte length of application output data area. If lenData
is 0 and pData is not null, it means the data buffer
length is unknown due to the request being submitted via
an old IOCTL or DOSDEVIOTL interface.
```

#### 2.1.8.11.26 FS\_MKDIR - Make Subdirectory:

Purpose Create the specified directory.

```
int far pascal FS_MKDIR (pcdfsi, pcdfsd,
 pname, iCurDirEnd,
 pEABuf)
struct cdfsi far * pcdfsi;
struct cdfsd far * pcdfsd;
char far * pname;
unsigned short iCurDirEnd;
char far * pEABuf;
```

#### Where

pcdfsi pointer to file-system independent working directory structure.

pcdfsd pointer to file-system dependent working directory structure.

\* pname pointer to asciiz name of directory to be created. The  
\* FSD does not need to verify this pointer.

\* iCurDirEnd index of the end of the current directory in pname.  
\* This is used to optimize FSD path processing. If  
\* iCurDirEnd == -1 there is no current directory relevant  
\* to the name text, i.e. a device.

pEABuf pointer to extended attribute buffer. This buffer  
contains attributes that will be set upon creation of  
the new directory. If NULL, no extended attributes are  
to be set. Addressing of this data area has not been  
validated by the kernel (See FSH\_PROBEBUF).

Remarks The FSD needs to do the time stamping itself; there is no aid in the kernel for time stamping sub-directories. FAT only supports creation time stamp and sets the other two fields to zeros. An FSD should do the same. The FSD can obtain the current time/date from the infoseg.

A new directory called pname should be created if possible. The standard directory entries '.' and '..' should be put into the directory.

#### 2.1.8.11.27 FS\_MOUNT - Mount/Unmount volumes:

Purpose Examination of a volume by an FSD to see if it recognizes the file system format.

```
int far pascal FS_MOUNT (flag, pvpfsi, pvpfsd, hVPB, pBoot)
unsigned short flag;
struct vpfsi far * pvpfsi;
struct vpfsd far * pvpfsd;
unsigned short hVPB;
char far * pBoot;
```

#### Where

flag indicates operation requested.

flag == 0 indicates that the FSD is requested to mount or accept a volume.

flag == 1 indicates that the FSD is being advised that the specified volume has been removed.

flag == 2 indicates that the FSD is requested to release all internal storage assigned to that volume as it has been removed from its drive and the last kernel-managed reference to that volume has been removed.

flag == 3 indicates that the FSD is requested to accept the volume regardless of recognition in preparation for formatting for use with the FSD.

All other values reserved. The value passed to the FSD will be valid.

pvpfsi pointer to file-system-independent portion of VPB. If the media contains an OS/2-recognizable boot sector, then the vpi\_vid field contains the 32-bit identifier for that volume. If the media does not contain such a boot sector, the FSD must generate a unique label for the media and place it into the vpi\_vid field.

pvpfsd pointer to file-system-dependent portion of VPB. The FSD may store information as necessary into this area.

hVPB handle to volume.

pBoot pointer to sector 0 read from the media. This pointer is ONLY valid when flag == 0. The buffer the pointer

refers to MUST NOT BE MODIFIED. The pointer is always valid and does not need to be verified when flag == 0; if a read error occurred, the buffer will contain zeroes.

Remarks The FSD should examine the volume presented and determine whether it recognizes the file system. If so, it should return zero after having filled in appropriate parts of vpfsi and vpfsd. The vpi\_vid and vpi\_text fields must be filled in by the FSD. If the FSD has an OS/2 format boot sector, it must convert the label from the media into asciiz form. The vpi\_hDev field will be filled in by OS/2. If the volume is unrecognized, the driver should return non-zero.

The vpi\_text and vpi\_vid must be updated by the FSD each time these values change.

The contents of the vpfsd are as follows:

(FLAG = 0)

The FSD is expected to issue an FSD\_FINDDUPHVPB to see if a duplicate VPB exists if one does exist the VPB fs dependent area of the new VPB is invalid and the new VPB will be unmounted after the FSD returns from the MOUNT. The FSD is expected to update the fs dependent area of the old duplicate VPB.

If no duplicate VPB exists the FSD should initialize the fs dependent.

(FLAG = 1)

VPB fs dependent part is same as when FSD last modified it.

(FLAG = 2)

VPB fs dependent part is same as when FSD last modified it.

After media recognition time, the volume parameters may be examined using the FSH\_GETVOLPARM call. The volume parameters should not be changed after media recognition time.

During a mount request, the FSD may examine other sectors on the media by using FSH\_DOVOLIO to perform the I/O. If an uncertain-media return is detected, the FSD is expected to clean up and return ERROR\_UNCERTAIN\_MEDIA in order to allow the volume mount logic to restart on the newly-inserted media. The FSD must provide the buffer to use for additional I/O.

The kernel manages the VPB via a ref count. All volume-specific objects are labelled with the appropriate volume handle and

represent references to the VPB. When all kernel references to a volume disappear, FS\_MOUNT is called with flag == 2, indicating a dismount request.

When the kernel detects that a volume has been removed from its drive, but there are still outstanding references to the volume, FS\_MOUNT is called with flag == 1 to allow the FSD to drop clean (or other regenerable) data for the volume. Data which is dirty and cannot be regenerated should be kept so that it may be written to the volume when it is remounted in the drive.

When a volume is to be formatted for use with an FSD, the kernel calls the FSD's FS\_MOUNT entry with flag == 3 to allow the FSD to prepare for the format operation. The FSD should accept the volume even if it is not a volume of the type that FSD recognizes, since the point of format is to change the filesystem on the volume. The operation may be failed if formatting doesn't make any sense. (For example, an FSD which supports only CD-ROM.)

Since the hardware does not allow for kernel-mediated removal of media, it is certain that the unmount request is issued when the volume is not present in any drive.

# 2.1.8.11.28 FS\_MOVE - Move a File or Subdirectory:

Purpose Moves (renames) the specified file or subdirectory.

```
int far pascal FS_MOVE (pcdfsi, pcdfsd,
 pSrc, iSrcCurDirEnd,
 pDst, iDstCurDirEnd)
```

```
struct cdfsi far * pcdfsi;
struct cdfsd far * pcdfsd;
char far * pSrc;
unsigned short iSrcCurDirEnd;
char far * pDst;
unsigned short iDstCurDirEnd;
```

## Where

pcdfsi pointer to file-system independent working directory structure.

pcdfsd pointer to file-system dependent working directory structure.

\* pSrc pointer to asciiz name of source file/directory. The FSD does not need to verify this pointer.

\* iSrcCurDirEnd index of the end of the current directory in pSrc. This is used to optimize FSD path processing. If iSrcCurDirEnd == -1 there is no current directory relevant to the source name text.

\* pDst pointer to asciiz name of destination file/directory. The FSD does not need to verify this pointer.

iDstCurDirEnd index of the end of the current directory in pDst. This is used to optimize FSD path processing. If iDstCurDirEnd == -1 there is no current directory relevant to the destination name text.

Remarks The file specified in filename should be moved or renamed to be destfile if possible.

\* Neither the source nor the destination filename may contain wildcard characters.

+ In the case of a Subdirectory move, system does the following checking:

+ \* No files in this directory or its sub-directories are open.

+ \* This directory or any of its sub-directories is not the current directory for any process in the system.

+ In addition, system also checks for circularity in source and target directory names; i.e. the source directory is not a prefix of the target directory.

+ Reminder: OS/2 does not validate input parameters, so FSD should call FSH\_PROBEBUF where appropriate.



#### 2.1.8.11.29 FS\_NEWSIZE - Change File's Logical Size:

**Purpose** Changes a file's logical (EOD) size.

```
int far pascal FS_NEWSIZE (psffsi, psffsd, len, IOflag)
struct sffsi far * psffsi;
struct sffsd far * psffsd;
unsigned long len;
unsigned short IOflag;
```

**Where**

psffsi pointer to file-system independent portion of open file instance.

psffsd pointer to file-system dependent portion of open file instance.

len desired new length of file.

IOflag indicates information about the operation on the handle.

IOflag == 0x0010 indicates write-through.

IOflag == 0x0020 indicates no-cache.

**Remarks** The FSD should return an error if an attempt is made to set the size to beyond the end of the direct access device.

The file system driver should attempt to set the size (EOD) of the file to newsize and update sfi\_size if successful. If the new size is larger than the currently allocated size, the file system driver should to the extent possible arrange for efficient access to the newly allocated storage.

Of the information passed in IOflag, the write-through bit is a MANDATORY bit in that any data written to the block device must be put out on the medium before the device driver returns. The no-cache bit, on the other hand, is an advisory bit that says whether the data being transferred is worth caching or not.

#### 2.1.8.11.30 FS\_NMPIPE - Do a remote named pipe operation:

**Purpose** Perform a special purpose named pipe operation remotely.

```
int far pascal FS_NMPIPE(psffsi, psffsd, OpType, pOpRec,
 pData, pName);
struct sffsi far * psffsi;
struct sffsd far * psffsd;
unsigned short OpType;
union npper far * pOpRec;
char far * pData;
char far * pName;
```

**Where**

psffsi pointer to file-system independent portion of open file instance.

psffsd pointer to file-system dependent portion of open file instance.

OpType operation to be performed. This parameter will take on the following values:

|                     |      |
|---------------------|------|
| NMP_GetPHandState   | 0x21 |
| NMP_SetPHandState   | 0x01 |
| NMP_PipeQInfo       | 0x22 |
| NMP_PeekPipe        | 0x23 |
| NMP_ConnectPipe     | 0x24 |
| NMP_DisconnectPipe  | 0x25 |
| NMP_TransactPipe    | 0x26 |
| NMP_READRAW         | 0x11 |
| NMP_WRITERAW        | 0x31 |
| NMP_WAITPIPE        | 0x53 |
| NMP_CALLPIPE        | 0x54 |
| NMP_QNmPipeSemState | 0x58 |

pOpRec data record which varies depending on the value of OpType. The first parameter in each structure encodes the length of the parameter block. The second parameter if non-zero indicates that the pData parameter is supplied and gives its length. The following record formats are used:

```
union npoper {
 struct phs_param phs;
 struct npi_param npi;
 struct npr_param npr;
 struct npw_param npw;
 struct npq_param npq;
 struct npx_param npx;
 struct npp_param npp;
 struct npt_param npt;
 struct qnps_param qnps;
 struct npc_param npc;
 struct npd_param npd;
}; /* npoper */

/* Get/SetPHandState parameter block */
struct phs_param {
 short phs_len;
 short phs_dlen;
 short phs_pmode; /* pipe mode set or returned */
};

/* DosQNmPipeInfo parameter block,
 * data is info. buffer addr */
struct npi_param {
 short npi_len;
 short npi_dlen;
 short npi_level; /* information level desired */
};

/* DosRawReadNmPipe parameters,
 * data is buffer addr */
struct npr_param {
 short npr_len;
 short npr_dlen;
 short npr_nbyt; /* number of bytes read */
};

/* DosRawWriteNmPipe parameters,
 * data is buffer addr */
struct npw_param {
 short npw_len;
 short npw_dlen;
 short npw_nbyt; /* number of bytes written */
};
```

```
/* NPipeWait parameters */
struct npq_param {
 short npq_len;
 short npq_dlen;
 long npq_timeo; /* timeout in milliseconds */
 short npq_prio; /* priority of caller */
};

/* DosCallNmPipe parameters,
 * data is in-buffer addr */
struct npx_param {
 short npx_len;
 unsigned short npx_ilen; /* length of in-buffer */
 char far *npx_obuf; /* pointer to out-buffer */
 unsigned short npx_olen; /* length of out-buffer */
 unsigned short npx_nbyt; /* number of bytes read */
 long npx_timeo; /* timeout in milliseconds */
};

/* PeekPipe parameters, data is buffer addr */
struct npp_param {
 short npp_len;
 unsigned short npp_dlen;
 unsigned short npp_nbyt; /* number of bytes read */
 unsigned short npp_avl0; /* bytes left in pipe */
 unsigned short npp_avl1; /* bytes left in current msg */
 unsigned short npp_state; /* pipe state */
};

/* DosTransactNmPipe parameters,
 * data is in-buffer addr */
struct npt_param {
 short npt_len;
 unsigned short npt_ilen; /* length of in-buffer */
 char far *npt_obuf; /* pointer to out-buffer */
 unsigned short npt_olen; /* length of out-buffer */
 unsigned short npt_nbyt; /* number of bytes read */
};
```

```
/* QNmpipeSemState parameter block,
 * data is user data buffer */
struct qnps_param {
 unsigned short qnps_len; /* length of parameter block */
 unsigned short qnps_dlen; /* length of supplied data block */
 long qnps_semh; /* system semaphore handle */
 unsigned short qnps_nbyt; /* number of bytes returned */
};
```

```
/* ConnectPipe parameter block, no data block */
struct npc_param {
 unsigned short npc_len; /* length of parameter block */
 unsigned short npc_dlen; /* length of data block */
};
```

```
/* DisconnectPipe parameter block, no data block */
struct npd_param {
 unsigned short npd_len; /* length of parameter block */
 unsigned short npd_dlen; /* length of data block */
};
```

pData pointer to user data buffer for operations which require it. When the pointer is supplied, its length will be given by the second element of the pOpRec structure.

pName pointer to remote pipe name. Supplied only for NMP\_WAITPIPE and NMP\_CALLPIPE operations. For these two operations only, the psffsi and psffsd parameters have no significance.

Remarks This entry point is for support of special remote named pipe operations. Not all pointer parameters are used for all operations. In cases where a particular pointer has no significance, it will be NULL.

This entry point will be called only for the UNC FSD. Non-UNC FSDs are required to have this entry point, but should return "not supported" if called.

#### 2.1.8.11.31 FS\_OPENCREATE - Open a File:

Purpose Opens (or creates) the specified file.

```
int far pascal FS_OPENCREATE (pcdfsi, pcdfsd,
 pName, iCurDirEnd,
 psffsi, psffsd,
 fhandflag, openflag,
 pAction, attr,
 pEABuf)
```

```
struct cdfsi far * pcdfsi;
struct cdfd far * pcdfsd;
char far * pName;
unsigned short iCurDirEnd;
struct sffsi far * psffsi;
struct sffsd far * psffsd;
unsigned short fhandflag;
unsigned short openflag;
unsigned short far * pAction;
unsigned short attr;
char far * pEABuf;
```

pcdfsi pointer to file-system independent working directory structure. The contents of this structure are invalid for direct access opens.

pcdfsd pointer to file-system dependent, working directory structure. The contents of this structure are invalid for direct access opens. For remote character devices, this field will contain a pointer to a DWORD that was obtained from the remote FSD when the remote device was attached to this FSD. The FSD can use this DWORD to identify the remote device if it wishes.

pName pointer to asciiz name of file to be opened. The FSD does not need to verify this pointer.

iCurDirEnd index of the end of the current directory in pName. This is used to optimize FSD path processing. If iCurDirEnd == -1 there is no current directory relevant to the name text, i.e. a device. This value is invalid for direct access opens.

psffsi pointer to file-system independent portion of open file instance.

psffsd pointer to file-system dependent portion of open file instance.

**fhandflag** indicates the desired sharing mode and access mode for the file handle. See the API documentation for the **OpenMode** parameter for **DOSOPEN**. An additional access mode 3 is defined when the file is being opened on behalf of the OS/2 loaded for purposes of executing a file or loading a module. If the file system does not support an executable attribute, it should treat this access mode as open for reading. The value of **fhandflag** passed to the FSD will be valid.

**openflag** indicates the action taken when the file is present or absent. See the API documentation for the **OpenFlag** for **DOSOPEN**. The value of **openflag** passed to the FSD will be valid. This value is invalid for direct access opens.

**pAction** location where FSD returns a description of the action taken as governed by **openflag**. The FSD does not need to verify this pointer. The contents of **Action** are invalid on return for direct access opens.

**attr** OS/2 file attributes. This value is invalid for direct access opens.

**pEABuf** pointer to extended attribute buffer. This buffer contains attributes that will be set upon creation of a new file or upon replacement of an existing file. If **NULL**, no extended attributes are to be set. Addressing of this data area has not been validated by the kernel (See **FSH\_PROBEBUF**). The contents of **EABuf** are invalid on return for direct access opens.

**Remarks** For the file create operation, if successful, **ST\_SCREAT** and **ST\_PCREAT** are set. This will cause the file to have 0 as last read and last write time. If it is desirable to make the last read/write time stamps same as the create time, simply set **ST\_SWRITE**, **ST\_PWRITE**, **ST\_SREAD**, and **ST\_PREAD** as well.

For the file open operation, the FSD should copy all supported time stamps from the directory entry into the SFT.

The sharing mode may be zero if this is a request to open a file from the 3.x box in compatibility mode or for an FCB request.

FCB requests for read-write access to a read-only file should be mapped to read-only access and reflected in the **sfi\_mode** field by the FSD. An FCB request is indicated by the third bit set in the **sfi\_type** field.

The flags defined for the **sfi\_type** field are:

**type == 0x0000** indicates file.

**type == 0x0001** indicates device.

**type == 0x0002** indicates named pipe.

**type == 0x0004** indicates FCB open.

All other values reserved.

FSDs are required to initialize the **sfi\_type** field, preserving the FCB bit.

Also on entry, the **sfi\_hvpb** field is filled in. If the file's logical size (EOD) is specified, it will be passed in the **sfi\_size** field. To the extent possible, the file system should try to allocate this much storage for efficient access.

Extended attributes are set for 1) the creation of a new file, 2) the truncation of an existing file, and 3) the replacement of an existing file. They are not set for a normal open.

If the standard OS/2 file creation attributes have been specified, they will be passed in the **attr** field. To the extent possible, the file system should interpret the extended attributes and apply them to the newly created or existing file. Extended attributes (EAs) that the file system does not itself use should be retained with the file and not discarded or rejected.

FSDs are required to support direct access opens. These are indicated by a bit set in the **sffsi.sfi\_mode** field. See **DOSOPEN** for information. Some of the parameters passed to the FSD for direct access opens are invalid, as described above.

On a successful return, the following fields in the **sffsi** structure must be filled in by the file system driver: **sfi\_size** and all the time and date fields.

The file-system dependent portion of open file instance passed to the FSD for **FS\_OPENCREATE** will always be uninitialized.

Infinite FCB opens of the same file by the same 3xbox process is supported. The first open is passed through to the FSD. Subsequent opens are not seen by the FSD.

Any non-zero value returned by the FSD indicates that the open failed and the file is not open.

\* 2.1.8.11.32 FS\_PATHINFO - Query/Set a File's Information:

\* Purpose Returns information for a specific path or file.

```
*
* int far pascal FS_PATHINFO (flag, pcdfsi, pcdfsd,
* pname, iCurDirEnd,
* level, pData, cbData)
*
* unsigned short flag;
* struct cdfs_i far * pcdfsi;
* struct cdfs_d far * pcdfsd;
* char far * pname;
* unsigned short iCurDirEnd;
* unsigned short level;
* char far * pData;
* unsigned short cbData;
```

Where

```
+ flag is a bit defined as follows:
+
+ flag == 0x0000 indicates retrieving information.
+
+ flag == 0x0001 indicates setting information on the
+ media.
+
+ flag == 0x0010 indicates that the information being
+ set must be written-through onto the disk before
+ returning. This bit is never set when retrieving
+ information.
+
+ All other values reserved.
+
+ pcdfsi pointer to file-system independent working directory
+ structure.
+
+ pcdfsd pointer to file-system dependent working directory
+ structure.
+
+ pname pointer to ascii name of file or directory for which
+ information is to be retrieved or set. The FSD does not
+ need to verify this pointer.
+
+ iCurDirEnd index of the end of the current directory in pname.
+ This is used to optimize FSD path processing. If
+ iCurDirEnd == -1 there is no current directory relevant
+ to the name text, i.e. a device.
+
+ level information level to be returned. Level selects among a
+ series of structures of data to be returned or set.
```

pData address of application data area. Addressing of this data area has not been validated by the kernel (See FSH\_PROBEBUF). When retrieval (flag == 0) is specified, the FSD will place the information into the buffer. When outputting information to a file (flag == 1), the FSD will retrieve that data from the application buffer.

cbData length of the application data area. For flag == 0, this is the length of the data the application wishes to retrieve. If there is not enough room for the entire level of data to be returned, the FSD will return ERROR\_BUFFER\_OVERFLOW. For flag == 1, this is the length of data to be applied to the file.

\* Remarks See DOSQPATHINFO and DOSSETPATHINFO for information level descriptions.

\* The FSD will not be called for DOSQPATHINFO level 5.

Microsoft Confidential  
OS/2 1.2 IFS Patent Documentation

\* 2.1.8.11.33 FS\_PROCESSNAME - Allow FSD to modify name after OS/2:  
\* canonicalization

\* Purpose Allow FSD to modify filename to its own specification after the  
\* OS/2 canonicalization process has completed.

```
int far pascal FS_PROCESSNAME (pNameBuf)
char far * pNameBuf;
```

\* Where

\* pNameBuf pointer to ASCII2 pathname. The FSD should modify the  
\* pathname in place. The buffer is guaranteed to be the  
\* length of the maximum path. The FSD does not need to  
\* verify this pointer.

\* Remarks The resulting name must be within the maximum path length returned  
\* by DOSQSYSINFO.

\* This routine allows the FSD to enforce a different naming  
\* convention than OS/2. For example, an FSD could remove blanks  
\* embedded in component names or return an error if it found such  
\* blanks. It is called after the OS/2 canonicalization process has  
\* succeeded. It is not called for FSH\_CANONICALIZE.

| This routine will be called for all APIs that use pathnames.

| This routine must return no error if the function is not  
| supported.

Microsoft Confidential  
OS/2 1.2 IFS Patent Documentation

2.1.8.11.34 FS\_READ - Read from a File:

Purpose Read the specified number of bytes from a file to a buffer  
location.

```
int far pascal FS_READ (psffsi, psffsd, pData, pLen, IOflag)
struct sffsi far * psffsi;
struct sffsd far * psffsd;
char far * pData;
unsigned short far * pLen;
unsigned short IOflag;
```

Where

psffsi pointer to file-system independent portion of open file  
instance. sfi\_position is the location within the file  
where the data is to be read from. The FSD should  
update the sfi\_position field.

psffsd pointer to file-system dependent portion of open file  
instance.

pData address of application data area. Addressing of this  
data area has not been validated by the kernel (See  
FSH\_PROBEBUF).

pLen pointer to length of the application data area. On  
input, this is the number of bytes that to be read. On  
output, this is the number of byte successfully read.  
If the application data area is smaller than the length,  
no transfer is to take place. The FSD will not be  
called for zero length reads. The FSD does not need to  
verify this pointer.

IOflag indicates information about the operation on the handle.

IOflag == 0x0010 indicates write-through.

IOflag == 0x0020 indicates no-cache.

Remarks If read is successful and is a file, the FSD should set  
ST\_SREAD and ST\_PREAD to make the kernel time stamp last  
modification time in the SFT.

Of the information passed in IOflag, the write-through  
bit is a MANDATORY bit in that any data written to the  
block device must be put out on the medium before the

device driver returns. The no-cache bit, on the other hand, is an advisory bit that says whether the data being transferred is worth caching or not.

#### 2.1.8.11.35 FS\_RMDIR - Remove Subdirectory:

**Purpose** Removes a subdirectory from the specified disk.

```
int far pascal FS_RMDIR (pcdfsi, pcdfsd,
 pname, iCurDirEnd)
struct cdfsi far * pcdfsi;
struct cdfsd far * pcdfsd;
char far * pname;
unsigned short iCurDirEnd;
```

**Where**

pcdfsi pointer to file-system independent working directory structure.

pcdfs d pointer to file-system dependent working directory structure.

\* pname pointer to asciiz name of directory to be removed. The  
\* FSD does not need to verify this pointer.

\* iCurDirEnd index of the end of the current directory in pname.  
\* This is used to optimize FSD path processing. If  
\* iCurDirEnd == -1 there is no current directory relevant  
\* to the name text, i.e. a device.

**Remarks** OS/2 has already assured that the directory being removed is not the current directory nor the parent of any current directory of any process.

The FSD should not remove any directory that has entries other than . and .. in it.

EP 0 415 346 A2

\* 2.1.8.11.36 FS\_SETSWAP - Notification of swap-file ownership:

Purpose Perform whatever actions are necessary to support the swapper.

```
int far pascal FS_SETSWAP (psffsi, psffsd)
struct sffsi far * psffsi;
struct sffsd far * psffsd;
```

Where

psffsi pointer to file-system independent portion of open file instance of the swapper file.

psffsd pointer to file-system dependent portion of open file instance.

\* Note Swapping will not begin until this calls returns successfully. This call will be made during system initialization.

\* The FSD will make all segments that are relevant to swap-file I/O non-swappable (see FSH\_FORCENOSWAP). This includes any data and code segments accessed during a read or write.

\* Any FSD that manages writeable media may be the swapper file system.

\* FS\_SETSWAP may be called more than once for the same or different volumes or FSDs. FSDs should be prepared to see multiple swapping files on more than one volume in future releases of OS/2.

+ 2.1.8.11.37 FS\_SHUTDOWN - Shutdown File System for Power Off:

+ Purpose Used to shutdown an FSD in preparation for power-off or IPL.

```
+ int far pascal FS_SHUTDOWN (type, reserved)
+ unsigned short type;
+ unsigned long reserved;
```

+ Where

+ type indicates what type of a shutdown operation to perform.

+ type == 0 indicates that the shutdown sequence is beginning. The kernel will not allow any new IO calls to reach the FSD. The only exception will be IO to the swap file by the swap thread through the FS\_READ and FS\_WRITE entry points. The kernel will still allow any thread to call FS\_COMMIT, FS\_FLUSHBUF and FS\_SHUTDOWN. The FSD should complete all pending calls that might generate disk corruption.

+ type == 1 indicates that the shutdown sequence is ending. An FS\_COMMIT has been called on every SFT open on the FSD and following that an FS\_FLUSHBUF on all volumes has been called. All final clean up activity must be completed before this call returns.

+ reserved reserved for future expansion.

+ Remarks From the perspective of an FSD, the shutdown sequence looks like this:

+ First, the system will call the FSD's FS\_SHUTDOWN entry with type == 0. This notifies the FSD that the system will begin committing SFTs in preparation for system power off. The kernel will not allow any new IO calls to the FSD once it receives this first call, except from the swapper thread. The swapper thread will continue to call the FS\_READ and FS\_WRITE entry points to read and write the swap file. The swapper thread will not attempt to grow or shrink the swap file nor should the FSD reallocate it. The kernel will continue to allow FS\_COMMIT and FS\_FLUSHBUF calls from any thread. This call should not return from the FSD until disk data modifying calls have completed to insure that a thread already inside the FSD does not wake and change disk data.

+ After the first FS\_SHUTDOWN call returns, the kernel will start committing SFTs. The FSD will see a commit for every SFT



associated with it. During these FS\_COMMIT calls, the FSD must flush any data associated with these SFTs to disk. The FSD must not allow any FS\_COMMIT or FS\_FLUSHBUF call to block permanently.

Once all of the SFTs associated with the FSD have been committed, FS\_FLUSHBUF will be called to flush all volumes. Following this, FS\_SHUTDOWN will be called with type == 1. This will tell the FSD to flush all buffers to disk. From this point, the FSD must not buffer any data destined for disk. Reads and writes to the swap file will continue, but the allocation of the swap file will not change. Once this call has completed, no file system corruption should occur if power is shut off.

#### 2.1.8.11.38 FS\_WRITE - Write to a File:

**Purpose** Write the specified number of bytes to a file from a buffer location.

```
int far pascal FS_WRITE (psffsi, psffsd,
 pData, pLen, IOflag)
struct sffsi far * psffsi;
struct sffsd far * psffsd;
char far * pData;
unsigned short far * pLen;
unsigned short IOflag;
```

**Where**

**psffsi** pointer to file-system independent portion of open file instance. sfi\_position is the location within the file where the data is to be written. The FSD should update the sfi\_position and sfi\_size fields.

**psffsd** pointer to file-system dependent portion of open file instance.

**pData** address of application data area. Addressing of this data area has not been validated by the kernel (See FSH\_PROBEBUF).

**pLen** pointer to length of the application data area. On input, this is the number of bytes that to be written. On output, this is the number of byte successfully written. If the application data area is smaller than the length, no transfer is to take place. The FSD does not need to verify this pointer.

**IOflag** indicates information about the operation on the handle.

IOflag == 0x0010 indicates write-through.

IOflag == 0x0020 indicates no-cache.

**Remark** If write is successful and is a file, the FSD should set ST\_SWRITE and ST\_PWRITE to make the kernel time stamp last modification time in the SFT.

The FSD should return an error for a direct access handle, if the write is beyond the end.

Of the information passed in IOflag, the write-through bit is a MANDATORY bit in that any data written to the block device must be

EP 0 415 346 A2

put out on the medium before the device driver returns. The no-cache bit, on the other hand, is an advisory bit that says whether the data being transferred is worth caching or not.

#### 2.1.8.12 FS Helper Functions

##### 2.1.8.12.1 FS Services Supplied By OS/2:

An FSD may access OS/2 services via two different paths: via DOS API calls at initialization time, via dynlinks to OS/2 FS-help services at task time.

The set of DOS API calls available at initialization time is equivalent to those available to device drivers at initialization time. (See IPL section near front of spec.) This includes DosOpen, DosRead, DosWrite, DosDevIoctl and DosClose among others.

##### 2.1.8.12.2 FS Help Routines:

FSDs are loaded as dynlink libraries and are able to import services provided by the kernel. These services can be called directly by the file system, passing the relevant parameters.

No validation of input parameters is done unless otherwise specified. The FSD should call FSH\_PROBEBUF where appropriate before calling the FS help routine.

When any service returns an error code, the FSD must wind out of the particular FS call as soon as possible and return the specific error code from the helper to the FS router.

There are many deadlocks that may occur as a result of operations issued by FSDs. OS/2 provide no means whereby deadlocks between file systems and applications can be detected.

The FSD helper routines are:

FSH\_ADDSHARE Add a name to the sharing set

FSH\_BUFSTATE Get or set buffer state

FSH\_CANONICALIZE Convert pathname to canonical form

FSH\_CHECKEANAME Check extended attribute name validity

FSH\_CRITERORR Signal a hard error to the daemon

\* FSH\_DEVIOCTL Send IOCTL request to device driver

\* FSH\_DOVOLIO Volume-based sector-oriented transfer

\* FSH\_DOVOLIO2 Send volume-based IOCTL request to device driver

\* FSH\_FINDCHAR Find first occurrence of char in string

\* FSH\_FINDDUPHVPB Locates equivalent hVPBs

\* FSH\_FLUSHBUF Flush buffered data to a volume

\* FSH\_FORCENOSWAP Force segments permanently into memory

\* FSH\_GETBUF Buffered sector read

\* FSH\_GETFIRSTOVERLAPBUF Locates first buffer overlapping range

FSH\_GETVOLPARM Get VPB data from VPB handle

FSH\_INTERR Signal an internal error

FSH\_ISCURDIRPREFIX Test for a prefix of a current directory

FSH\_LOADCHAR Load character from a string

FSH\_PREVCHAR Move backward in string

\* FSH\_PROBEBUF User address validity check

\* FSH\_QSYSINFO Query system information

FSH\_RELEASEBUF Release the owned buffer

FSH\_REMOVESHARE Remove a name from the sharing set

FSH\_SEGALLOC Allocate a GDT or LDT segment

FSH\_SEGFREE Release a GDT or LDT segment

FSH\_SEGREALLOC Change segment size

FSH\_SEMCLEAR Request a semaphore

FSH\_SEMREQUEST Request a semaphore

FSH\_SEMSET Set a semaphore

FSH\_SEMSETWAIT Set a semaphore and wait for clear

FSH\_SEMWAIT Wait for clear

+ FSH\_STORECHAR Store character into string  
+ FSH\_UPPERCASE Uppercase ascii string  
FSH\_WILDMATCH Match using OS/2 wildcards  
FSH\_YIELD Yield CPU to higher priority threads

#### 2.1.8.12.3 FSH\_ADDSHARE - Add a name to the share set:

FSH\_ADDSHARE adds a name to the currently active sharing set. FSDs will use this call when executing deletes or renames with wildcard characters.  
\* FSH\_ADDSHARE returns a share handle that must be used with a succeeding FSH\_REMOVESHARE.

```
int far pascal FSH_ADDSHARE (pName, mode, hVPB, phShare)
char far * pName;
unsigned short mode;
unsigned short hVPB;
unsigned long far *phShare;
```

#### Where

|   |         |                                                                                                                                                                                 |
|---|---------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| * | pName   | pointer to ascii name to be added into the share set. The name must be in canonical form: no '.' or '..' components, uppercase, no metacharacters, and full pathname specified. |
| * | mode    | sharing mode and access mode as defined in the DOSOPEN API. All other bits (Direct Open, Write-Through, etc) must be zero.                                                      |
| * | hVPB    | handle to volume where the named object is presumed to exist.                                                                                                                   |
| * | phShare | pointer to location where a share handle is stored. This handle may be passed to FSH_REMOVESHARE.                                                                               |

Returns Error code if error detected, 0 otherwise.

ERROR\_SHARING\_VIOLATION - the file is open with a conflicting sharing/access mode.

ERROR\_TOO\_MANY\_OPEN\_FILES - there are too many files open at the present time.

\* ERROR\_SHARING\_BUFFER\_EXCEEDED - there is not enough memory to hold sharing information.

\* ERROR\_INVALID\_PARAMETER - invalid bits in mode.

\* ERROR\_FILENAME\_EXCED\_RANGE - pathname is too long.

\* Note Do not call FSH\_ADDSHARE for character devices.

\* FSH\_ADDSHARE may block.

To help avoid deadlocks, FSH\_ADDSHARE should not be called while owning an OS/2 buffer.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH\_PROBEBUF where appropriate.

\* 2.1.8.12.4 FSH\_BUFSTATE - Get or set buffer state:

\* FSH\_BUFSTATE is used retrieve or set the state of a buffer.

```
* int far pascal FSH_BUFSTATE (pBuf, flag, pState)
* char far * pBuf;
* unsigned short flag;
* unsigned short far * pState;
```

\* Where

\* pBuf pointer to buffer. This pointer was returned to the FSD by FSH\_GETBUFFER

\* flag indicates get or set buf state.

\* flag == 0 indicates retrieve state

\* flag == 1 indicates set state

\* pState for set, points to new state of buffer on input. For retrieve, points to state of buffer.

\* State == 0x0080 indicates dirty buffer.

\* All other values are reserved.

\* Returns Error code if error detected, 0 otherwise.

\* ERROR\_INVALID\_PARAMETER - pointer to buffer is invalid or reserved state bits are set.

| Note To set the opposite of a defined state, call FSH\_BUFSTATE with the bit not set. For example, to set a buffer not dirty, pass 0x00 as the new state.

\* Reminder: OS/2 does not validate input parameters, so FSD should call FSH\_PROBEBUF where appropriate.

EP 0 415 346 A2

\* 2.1.8.12.5 FSH\_CANONICALIZE - Convert pathname to canonical form:

\* FSH\_CANONICALIZE is used to convert a pathname to a canonical form by processing '..'s and '..'s, uppercasing, and prepending the current directory to non-absolute paths.

```
* int far pascal FSH_CANONICALIZE (pPathName, cbPathBuf, pPathBuf)
* char far * pPathName;
* unsigned short cbPathBuf;
* char far * pPathBuf;
```

\* Where

pPathName pointer to asciiz pathname to be canonicalized.

cbPathBuf length of pathname buffer.

pPathBuf pointer to buffer to copy canonicalized path into.

\* Returns Error code if error detected, 0 otherwise.

ERROR\_PATH\_NOT\_FOUND - invalid pathname - too many '..'s.

ERROR\_BUFFER\_OVERFLOW - pathname is too long.

\* Note This routine process DBCS characters properly.

The FSD is responsible for verifying the string pointers and checking for segment boundaries.

FSH\_CANONICALIZE should be called for names passed into the FSD raw data packets. For example, names passed to FS\_FSCtl in the parameter area should be passed to FSH\_CANONICALIZE. This routine does not need to be called for explicit names passed to the FSD, i.e., the name passed to FS\_OPENCREATE.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH\_PROBEBUF where appropriate.

+ 2.1.8.12.6 FSH\_CHECKEANAME - Check extended attribute name validity.:

+ FSH\_CHECKEANAME is used to check the extended attribute name for illegal characters, including wildcard characters, and correct length.

```
+ int far pascal FSH_CHECKEANAME (iLevel, cbEName, szEName)
+ unsigned short iLevel;
+ unsigned long cbEName;
+ char far * szEName;
```

+ Where

iLevel extended attributes name checking level.

iLevel == 0x0001 indicates OS/2 version 1.2 name checking.

cbEName length of extended attribute name, not including terminating NUL.

szEName extended attribute name to check for validity.

+ Returns Error code if error detected, 0 otherwise.

ERROR\_INVALID\_NAME - pathname contains invalid or wildcard characters or is too long.

ERROR\_INVALID\_PARAMETER - invalid level.

+ Note This routine process DBCS characters properly.

The set of invalid characters for EA names is the same as that for filenames. In v1.2, the maximum length of an EA name, not including terminating NUL, is 255 bytes. The minimum length is 1 byte.

The FSD is responsible for verifying the string pointers and checking for segment boundaries.

FSH\_CHECKEANAME should be called for extended attribute names passed to the FSD.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH\_PROBEBUF where appropriate.

#### 2.1.8.12.7 FSH\_CRITERROR - Signal a hard error to the daemon:

FSH\_CRITERROR is used to pass an error from the current thread to the hard error daemon, block waiting for a response, and return the response to the caller.

```
int far pascal FSH_CRITERROR (cbMessage, pMessage,
 nSubs, pSubs, fAllowed)
```

```
unsigned short cbMessage;
char far * pMessage;
unsigned short nSubs;
char far * pSubs;
unsigned short fAllowed;
```

#### Where

cbMessage length of message template

pMessage pointer to message template. This may contain replaceable parameters in the format used by the message retriever.

nSubs number of replaceable parameters.

pSubs pointer to replacement text. The replacement text is a packed set of ASCII strings.

fAllowed bit mask of allowed actions:

Bit 0x0001 on indicates FAIL allowed

Bit 0x0002 on indicates ABORT allowed

Bit 0x0004 on indicates RETRY allowed

Bit 0x0008 on indicates IGNORE allowed

Bit 0x0010 on indicates ACKNOWLEDGE only allowed

All other bits are reserved and must be zero. If bit 0x0010 is set, and any or some of bits 0x0001 to 0x0008 are set, bit 0x0010 will be ignored.

Returns Action to be taken:

0x0000 ignore

0x0001 retry

0x0003 fail

0x0004 continue

#### Note

If the user responds with an action that is not allowed, it is treated as FAIL. If FAIL is not allowed, it is treated as ABORT. ABORT is always allowed.

When ABORT is the final action, OS/2 does not return this as an indicator, only that FAIL was returned. The actual ABORT operation is generated when this thread of execution is about to return to user code,

The maximum length of the template is 128 bytes (including the NUL). The maximum length of the message with text substitutions is 512 bytes. The maximum number of substitutions is 9.

If any action other than retry is selected for a given hard error popup, then any subsequent popups (within the same API call) will be automatically failed; a popup will not be done.

This means that (except for retries) there can be at most one hard error popup per call to the FSD. And, if the kernel generates a popup, then the FSD cannot create one.

FSH\_CRITERROR will FAIL automatically if the user application has set autofail, or if a previous hard error has occurred.

FSH\_CRITERROR may block.

To help avoid deadlocks, FSH\_CRITERROR should not be called while owning an OS/2 buffer.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH\_PROBEBUF where appropriate.

2.1.8.12.0 FSH\_DEVIOCTL - Send IOCTL request to device driver:

FSDs call FSH\_DEVIOCTL to control device driver operation independently from I/O operations. This is typically in filtering DOSDEVIOCTL requests when passing the request on to the device driver.

```
int far pascal FSH_DEVIOCTL (flag, hDev, sfn, cat, func,
 pParm, cbParm, pData, cbData)
unsigned short flag;
unsigned long hDev;
unsigned short sfn;
unsigned short cat;
unsigned short func;
char far * pParm;
unsigned short cbParm;
char far * pData;
unsigned short cbData;
```

Where

flag indicates whether the FSD initiated the call or not.

IOflag == 0x0000 indicates that the FSD is just passing user pointers on to the helper.

IOflag == 0x0001 indicates that the FSD initiated the DevIOctl call as opposed to passing a DevIOctl that it had received.

All other bits are reserved and must be zero.

hDev device handle obtained from VPB

sfn system file number from open instance that caused the FSH\_DEVIOCTL call. This field should be passed unchanged from the sfi selfsfn field. If no open instance corresponds to this call, this field should be set to 0xFFFF.

cat category of IOCTL to perform

func function within category of IOCTL.

pParm long address to parameter area

cbParm length of parameter area

pData long address to data area

cbData length of data area

Remarks An FSD needs to be careful of pointers to buffers that are passed to it from FS\_IOCTL, and what it passes to FSH\_DeVioCtl. It is possible that such pointers may be real mode pointers if the call was made from the 3x box. In any case, the FSD must indicate whether it initiated the DevIOctl call, in which case the kernel can assume that the pointers are all protected mode pointers, or if it is passing user pointers on the the FSH\_DeVioCtl call, in which case the mode of the pointers will depend on whether this is the 3x box or not. An important thing to note is that the FSD MUST NOT mix user pointers with its own pointers when using this helper.

Returns Error code if error detected, 0 otherwise.

ERROR\_INVALID\_FUNCTION - the function supplied is incompatible with the category supplied and the device handle supplied.

ERROR\_INVALID\_CATEGORY - the category supplied is incompatible with the function supplied and the device handle supplied.

Device driver error code

Note FSH\_DEVIOCTL may block.

To help avoid deadlocks, FSH\_DEVIOCTL should not be called while owning an OS/2 buffer.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH\_PROBEBUF where appropriate.

## 2.1.8.12.9 FSH\_DOVOLIO - Volume-based sector-oriented transfer:

FSH\_DOVOLIO performs I/O to the specified volume. It formats a device driver request packet for the requested I/O, locks the data transfer region, calls the device driver, and reports any errors to the hard error daemon before returning to the FSD. Any retries indicated by the hard error daemon or actions indicated by DOSERROR are done within the call to FSH\_DOVOLIO.

```
int far pascal FSH_DOVOLIO (operation, fAllowed, hVPB, pData, pcSec, iSec)
unsigned short operation;
unsigned short fAllowed;
unsigned short hVPB;
char far * pData;
unsigned short far * pcSec;
unsigned long iSec;
```

Where

| operation | bit | mask | indicating | read/read-bypass/write/<br>write-bypass/verify-after-write/write-through<br>and no-cache operation to be performed. |
|-----------|-----|------|------------|---------------------------------------------------------------------------------------------------------------------|
|-----------|-----|------|------------|---------------------------------------------------------------------------------------------------------------------|

Bit 0x0001 off indicates read.

Bit 0x0001 on indicates write.

Bit 0x0002 off indicates no bypass.

Bit 0x0002 on indicates cache bypass.

Bit 0x0004 off indicates no verify-after-write operation.

Bit 0x0004 on indicates verify-after-write.

Bit 0x0008 off indicates errors signalled to the hard error daemon.

Bit 0x0008 on indicates hard errors will be returned directly.

Bit 0x0010 off indicates I/O is not "write-through".

Bit 0x0010 on indicates I/O is "write-through".

Bit 0x0020 off indicates data for this I/O should probably be cached.

Bit 0x0020 on indicates data for this I/O should probably not be cached.

All other bits are reserved and must be zero.

The difference between the "cache bypass" and the "no cache" bits is in the type of request packet that the device driver will see. With "cache bypass", it will get a packet with command code 24, 25, or 26. With "no cache", it will get the extended packets for command codes 4, 8, or 9. The advantage of the latter is that the write-through bit can also be sent to the device driver in the same packet, thus improving the functionality at the level of the device driver.

fAllowed bit mask pf allowed actions:

Bit 0x0001 on indicates FAIL allowed

Bit 0x0002 on indicates ABORT allowed

Bit 0x0004 on indicates RETRY allowed

Bit 0x0008 on indicates IGNORE allowed

Bit 0x0010 on indicates ACKNOWLEDGE only allowed

All other bits are reserved and must be zero. If bit 0x0010 is set, none of the other bits may be set.

hVPB volume handle for source of I/O

pData long address of user transfer area

pcSec pointer to number of sectors to be transferred. On return this is the number of sectors successfully transferred.

iSec sector number of first sector of transfer

Returns Error code if operation failed, 0 otherwise.

ERROR\_PROTECTION\_VIOLATION - the supplied address/length is not valid.

ERROR\_UNCERTAIN\_MEDIA - the device driver can no longer reliably tell if the media has been changed. This occurs only within the context of an FS\_MOUNT call.

ERROR\_INVALID\_PARAMETER - invalid bits in fAllowed.

ERROR\_TRANSFER\_TOO\_LONG - transfer is too long for



device

Device-driver/device-manager errors listed "-----  
-----" on page ----.

Note FSH\_DOVOLIO may be used at all times within the FSD. When called within the scope of a FS\_MOUNT call, it applies to the volume in the drive without regard to which volume it may be. However, since volume recognition is NOT complete until the FSD returns to the FS\_MOUNT call, the FSD must take care when an ERROR\_UNCERTAIN\_MEDIA is returned. This indicates that the media has gone uncertain while we are trying to identify the media in the drive. This may indicate that the volume that the FSD was trying to recognize was removed. In this case, the FSD must release any resources attached to the hVPB passed in the FS\_MOUNT call and return ERROR\_UNCERTAIN\_MEDIA to the FS\_MOUNT call. This will direct the volume tracking logic to restart the mount process.

OS/2 will validate the user transfer area for proper access and size and will lock the segment.

If an error occurs during the transfer and an OS/2 buffer is owned, it will be released before signalling the hard error daemon.

Verify-after-write or write-through specified on a read will be ignored.

On 80386 processors, FSH\_DOVOLIO will take care of memory contiguity requirements of device drivers. It is advisable, hence, that FSDs use FSH\_DOVOLIO instead of calling device drivers directly. This will improve performance of FSDs running on 80386 processors.

FSH\_DOVOLIO may block.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH\_PROBEBUF where appropriate.

\* 2.1.8.12.10 FSH\_DOVOLIO2 - Send volume-based IOCTL request to device driver:

\* FSDs call FSH\_DOVOLIO2 to control device driver operation independently from  
\* I/O operations. This routine supports volume management for IOCTL  
\* operations. Any errors are reported to the hard error daemon before  
\* returning to the FSD. Any retries indicated by the hard error daemon or  
\* actions indicated by DOSERROR are done within the call to FSH\_DOVOLIO2.

\* int far pascal FSH\_DOVOLIO2 (hDev, sfn, cat, func, pParm,  
| cbParm, pData, cbData)

\* unsigned long hDev;  
\* unsigned short sfn;  
\* unsigned short cat;  
\* unsigned short func;  
\* char far \* pParm;  
\* unsigned short cbParm;  
\* char far \* pData;  
\* unsigned short cbData;

\* Where

\* hDev device handle obtained from VPB

\* sfn system file number from open instance that caused the  
FSH\_DEVIOCTL call. This field should be passed  
unchanged from the sfi\_selfsfn field. If no open  
instance corresponds to this call, this field should be  
set to 0xFFFF.

\* cat category of IOCTL to perform

\* func function within category of IOCTL

\* pParm long address to parameter area

\* cbParm length of parameter area

\* pData long address to data area

\* cbData length of data area

\* Returns Error code if error detected, 0 otherwise.

\* ERROR\_INVALID\_FUNCTION - the function supplied is incompatible  
\* with the category supplied and the device handle supplied.

\* ERROR\_INVALID\_CATEGORY - the category supplied is incompatible  
\* with the function supplied and the device handle supplied.

Device driver error code

Note The purpose of this routine is to enable volume tracking with ioctl's. It is not available at the API level.

FSH\_DOVOLIO2 may block.

To help avoid deadlocks, FSH\_DOVOLIO2 should not be called while owning an OS/2 buffer.

System does normal volume checking for this request.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH\_PROBEBUF where appropriate.

\* 2.1.8.12.11 FSH\_FINDCHAR - Find first occurrence of char in string:

\* Provides mechanism for find the first occurrence of any one of a set of characters in an ASCIIZ string taking into account DBCS considerations.

\* int far pascal FSH\_FINDCHAR (nChars, pChars, ppStr)  
\* unsigned short nChars;  
\* char far \* pChars;  
\* char far \* far \* ppStr;

\* Where

\* nChars number of chars in search list.

\* pChars array of chars to search for. These cannot be DBCS characters. The NULL character cannot be searched for.

\* ppStr pointer to character pointer to begin search from. This pointer is updated upon return to point to the character found. This must be an ASCIIZ string.

\* Returns Error code if match failed, 0 otherwise.

\* ERROR\_CHAR\_NOT\_FOUND - none of the characters were found.

\* Note The search will continue until a matching character is found or the end of the string is found.

\* The FSD is responsible for verifying the string pointers and checking for segment boundaries.

\* Reminder: OS/2 does not validate input parameters, so FSD should call FSH\_PROBEBUF where appropriate.

EP 0 415 346 A2

#### 2.1.8.12.12 FSH\_FINDDUPHVPB - Locate equivalent hVPB:

Provides mechanism for identifying previous instance of a volume during the FS\_MOUNT process. When OS/2 is recognizing a volume, it calls the FSD to mount the volume. At this point, the FSD may elect to allocate storage and buffer data for that volume. The mount process will allocate a new VPB whenever the media becomes uncertain (the device driver recognizes that it can no longer be certain that the media is unchanged). This VPB cannot be collapsed with a previously allocated VPB (due to a reinsertion of media) until the FS\_MOUNT call returns. However, the previous VPB may have some cached data that must be updated from the media (the media may have been written while it was removed). FSH\_FINDDUPHVPB allows the FSD to find this previous occurrence of the volume in order to update the cached information for the old VPB. Note that the newly created VPB will be unmounted if there is another, older VPB for that volume.

```
int far pascal FSH_FINDDUPHVPB (hVPB, phVPB)
unsigned short hVPB;
unsigned short far * phVPB;
```

##### Where

hVPB        handle to the volume to be found

phVPB      pointer to where handle of matching volume will be stored.

Returns    Error code if no matching VPB found. 0 otherwise.

ERROR\_NO\_ITEMS - there is no matching hVPB.

\* Note      Reminder: OS/2 does not validate input parameters, so FSD should call FSH\_PROBEBUF where appropriate.

#### 2.1.8.12.13 FSH\_FLUSHBUF - Flush buffered data to a volume:

Flush buffer takes dirty sectors contained in the buffer cache on a particular media and writes them out. Optionally, the data can be discarded afterwards.

```
int far pascal FSH_FLUSHBUF (hVPB, fDiscard)
unsigned short hVPB;
unsigned short fDiscard;
```

##### Where

hVPB        handle to the volume to be flushed

fDiscard    indicates disposition of cached data.

fDiscard == 0 indicates don't discard any buffers.

fDiscard == 1 indicates discard clean buffers

All other values are reserved.

Returns    Error code if any write failed. 0 otherwise.

ERROR\_INVALID\_PARAMETER - the value of Operation is invalid.

Device-driver/device-manager errors listed "-----" on page ---.

Note        If fDiscard = 1 and a write error occurred, the data in the buffer(s) that generated the error is not discarded.

\* See note under FSH\_GETBUF for interactions with other buffer calls.

\* Reminder: OS/2 does not validate input parameters, so FSD should call FSH\_PROBEBUF where appropriate.

FSH\_FLUSHBUF may block.

#### 2.1.8.12.14 FSH\_FORCENOSWAP - Force segments permanently into memory:

An FSD may call FSH\_FORCENOSWAP to force segments to be loaded into memory and marked non-swappable. All segments both in the load image of the FSD and those allocated via FSH\_SEGALLOC are eligible to be marked. There is no way to undo the effect of FSH\_FORCENOSWAP.

If an FSD is notified that it is managing the swapping media, it should make this call for the necessary segments.

```
int far pascal FSH_FORCENOSWAP (sel)
unsigned short sel;
```

Where

sel selector that is to be made non-swappable.

Returns Error code if invalid selector.

\* ERROR\_INVALID\_ACCESS - the selector is invalid.

\* ERROR\_ACCESS\_DENIED - the selector is invalid or the selector belongs to another process.

\* ERROR\_DIRECT\_ACCESS\_HANDLE - the handle doesn't refer to a segment.

\* ERROR\_NOT\_ENOUGH\_MEMORY - not enough physical memory to make segment non-swappable.

\* ERROR\_SWAP\_TABLE\_FULL, ERROR\_SWAP\_FILE\_FULL, ERROR\_PMM\_INSUFFICIENT\_MEMORY - attempt to grow the swap file failed.

\* Note FSH\_FORCENOSWAP may block.

\* Reminder: OS/2 does not validate input parameters, so FSD should call FSH\_PROBEBUF where appropriate.

#### \* 2.1.8.12.15 FSH\_GETBUF - Buffered sector read:

\* FSH\_GETBUF is used to access the OS/2 sector cache to retrieve a sector from a particular volume. The pointer to the OS/2 buffer cache element is returned. The data in the buffer may be pre-read if desired.

```
* int far pascal FSH_GETBUF (iSec, fPreRead, hVPB, ppBuf)
* unsigned long iSec;
* unsigned short fPreRead;
* unsigned short hVPB;
* char far * far * ppBuf;
```

\* Where

\* iSec sector number on the volume to return

\* fPreRead indicates whether the sector should be pre-read.

\* fPreRead == 0x0000 indicates pre-read sector.

\* fPreRead == 0x0001 indicates no pre-reading of sector.

\* All other values are reserved.

\* hVPB handle to the volume

\* ppBuf pointer to location where pointer to buffer data is returned

\* Returns Error code if operation failed, 0 otherwise.

\* ERROR\_GETBUF\_FAILED - the write to clear out a buffer failed or the read to fill the buffer in failed.

\* Note FSH\_GETFIRSTOVERLAPBUF, FSH\_GETBUF and FSH\_RELEASEBUF are used to obtain and release buffers. Any buffer that is owned by a thread is unavailable to all other threads until it is freed. At most one buffer may be owned by any thread.

Freeing occurs by calling FSH\_FLUSHBUF, FSH\_RELEASEBUF, or FSH\_GETBUF to retrieve another sector.

The buffer returned is marked as being owned by the calling thread. If any OS/2 buffer is still owned by an FSD when the system call completes, the system will halt with an Internal Error, thus the FSD is responsible for ensuring that all system owned buffers are released before it returns to the router.

Not being preread is a performance optimization for when a file is being grown and a partial sector is being filled in.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH\_PROBEBUF where appropriate.

FSH\_GETBUF may block.

#### 2.1.8.12.16 FSH\_GETFIRSTOVERLAPBUF - Locates buffer overlapping range:

Provides mechanism for handling buffers that overlap a large transfer region within a file. When performing large transfers, the FSD needs to identify all buffers that contain data that overlap a particular region of the media. When writing to the media, these overlapping buffers must be filled in with the new data as they contain old data while when reading these buffers need to be copied over the data transferred from the media.

Typically, when a large transfer is requested, the FSD will use an algorithm along the following lines:

```
isecFirst = lowest sector number to be transferred
isecLast = highest sector number to be transferred

/* While there are sectors still to be transferred
*/
while (isecFirst <= isecLast) {

 /* Find first buffer in cache that's in the transfer
 * range.
 */
 FSH_GETFIRSTOVERLAPBUF (hVPB, isecFirst, isecLast, &isecBuf,
 &pbuf);

 /* Perform the I/O to the first unbuffered part
 */
 csec = isecBuf - isecFirst;
 if (csec != 0)
 FSH_DOVOLIO (... , &csec, isecFirst);

 /* Handle the buffered data appropriately
 */
 ...

 /* Release the buffer
 */
 FSH_RELEASEBUF ();

 /* Reduce range of data-to-be-transferred
 */
 isecFirst = isecBuf + 1;

 /* Continue until no more sectors
 */
}
```

```
int far pascal FSH_GETFIRSTOVERLAPBUF (hVPB, isecFirst, isecLast,
 piseBuf, ppBuf)
```

```
unsigned short hVPB;
unsigned long isecFirst;
unsigned long isecLast;
unsigned long far * piseBuf;
char far * far * ppBuf;
```

Where

hVPB        handle for volume of I/O

isecFirst   logical sector number of beginning of range

isecLast    last logical sector number of range

piseBuf    pointer to returned logical sector number of buffered  
            sector

ppBuf      pointer to location where pointer to buffer data is  
            returned

Returns    Error code if no overlapping buffer found. 0 otherwise.

            ERROR\_NO\_ITEMS - no overlapping buffer was found.

Note       FSH\_GETFIRSTOVERLAPBUF may block.

            Reminder: OS/2 does not validate input parameters, so FSD should  
            call FSH\_PROBEBUF where appropriate.

            See note under FSH\_GETBUF for interactions with other buffer  
            calls.

2.1.8.12.17 FSH\_GETVOLPARM - Get VPB data from VPB handle:

FSH\_GETVOLPARM allows an FSD to retrieve file-system-independent and  
-dependent data from a VPB. Since the FS router passes in a VPB handle  
individual FSDs need to map the handle into pointers to the relevant  
portions.

```
void far pascal FSH_GETVOLPARM (hVPB, ppVPBfsi, ppVPBfsd)
unsigned short hVPB;
struct vpfsi far * far * ppVPBfsi;
struct vpfsd far * far * ppVPBfsd;
```

Where

hVPB        volume handle of interest,

ppVPBfsi    location of where pointer to file-system- independent  
            data is stored

ppVPBfsd    location of where pointer to file-system- dependent data  
            is stored

Returns    Nothing

Note       FSH\_GETVOLPARM will not block.

\*           Reminder: OS/2 does not validate input parameters, so FSD should  
\*           call FSH\_PROBEBUF where appropriate.

#### 2.1.8.12.18 FSH\_INTERR - Signal an internal error:

For reliability, if an FSD detects an internal inconsistency during normal operation, the safest thing is for the FSD to shut down the system as a whole, the reason being that it is not clear if the system as a whole is in a state that allows normal execution to continue.

When an FSD calls FSH\_INTERR, the address of the caller and the supplied message is displayed on the console. The system then halts.

```
void far pascal FSH_INTERR (pMsg, cbMsg)
```

```
char far * pMsg;
unsigned short cbMsg;
```

##### Where

pMsg        pointer to message text  
cbMsg       length of message text

Returns    None, does not return.

Note       The code used to display the message is primitive. The message should contain ASCII characters in the range 0x20-0x7E, optionally with 0x0D and 0x0A to break the text into multiple lines.

The FSD must preface all such messages with the name of the file system.

Maximum message length is 128 characters. Messages longer than this are truncated.

\*        Reminder: OS/2 does not validate input parameters, so FSD should  
\*        call FSH\_PROBEBUF where appropriate.

#### 2.1.8.12.19 FSH\_ISCURDIRPREFIX - Test for a prefix of a current directory:

Since the kernel manages the text of each current directory for each process, the FSD must disallow any modification of any directory that is either a current directory of some process or the parent of any current directory of some process. FSDs call FSH\_ISCURDIRPREFIX to achieve this.

\* FSH\_ISCURDIRPREFIX will take the supplied path name, enumerate all current  
\* directories in use and test to see if the specified path name is a prefix or  
\* is equal to some current directory.

```
int far pascal FSH_ISCURDIRPREFIX (pName)
char far *pName;
```

##### Where

\*        pName        pointer to path name. The name must be in canonical  
\*                       form: no '.' or '..' components, uppercase, no  
\*                       metacharacters, and full pathname specified.

Returns    Error code if pName is the prefix of or equal to the current directory of a process, 0 otherwise.

ERROR\_CURRENT\_DIRECTORY - the specified path is a prefix of or is equal to the current directory of some process.

\* Note       If the current directory is the root and the pathname is "d:\",  
\*               ERROR\_CURRENT\_DIRECTORY will be returned.

\*        FSH\_ISCURDIRPREFIX may block.

\*        Reminder: OS/2 does not validate input parameters, so FSD should  
\*        call FSH\_PROBEBUF where appropriate.

#### 2.1.8.12.20 FSH\_LOADCHAR - Load a character from a string:

Provides mechanism for loading a character from a string taking into account DBCS considerations.

```
void far pascal FSH_LOADCHAR (ppStr, pChar)
char far * far * ppStr;
unsigned short far * pChar;
```

##### Where

ppStr pointer to character pointer to string. The character at this location will be retrieved and this pointer will be updated.

pChar pointer to character returned. If character is non-DBCS, the first byte will be the character and the second byte will be zero.

Note Reminder: OS/2 does not validate input parameters, so FSD should call FSH\_PROBEBUF where appropriate.

#### 2.1.8.12.21 FSH\_NAMEFROMSFN - Get the full path name from an SFN.:

FSH\_NAMEFROMSFN allows an FSD to retrieve the full path name for an object to which an SFN refers.

```
void far pascal FSH_NAMEFROMSFN(sfn, pName, pcbName)
unsigned short sfn;
char far * pName;
unsigned short far * pcbName
```

##### Where

sfn system file number of file instance obtained from the sfi\_selfsfn field of the file system independent part of the SFT for the object.

pName location of where the returned full path name is to be stored.

pcbName location of where the FSD places the size of the buffer pointed to by pName. On return, the kernel will fill this in with the length of the path name. The length does not include the terminating null character. The size of the buffer should be long enough to hold the full path name, or else an error will be returned.

Returns Error code if the SFN is invalid, or the buffer is not big enough to hold the full path name. 0 if no error occurred.

ERROR\_INVALID\_HANDLE - the SFN is invalid.

ERROR\_BUFFER\_OVERFLOW - the buffer is too short for the returned path.

Note FSH\_NAMEFROMSFN will not block.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH\_PROBEBUF where appropriate.



#### 2.1.8.12.22 FSH\_PREVCHAR - Decrement character pointer:

Provides mechanism for decrementing a character pointer taking into account DBCS considerations.

```
void far pascal FSH_PREVCHAR (pBeg, ppStr)
char far * pBeg;
char far * far * ppStr;
```

Where

pBeg        pointer to beginning of string

ppStr      pointer to character pointer. This value is decremented appropriately upon return. If it is at the beginning of the string, the pointer is not decremented. If it points to the second byte of a DBCS char, it will be decremented to point to the first byte of the char.

Note       The FSD is responsible for verifying the string pointer and checking for segment boundaries.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH\_PROBEBUF where appropriate.

#### 2.1.8.12.23 FSH\_PROBEBUF - User address validity check:

Since users may pass in arbitrary pointers to data, FSDs must perform validity checks on these pointers before using them. Because OS/2 is multithreaded, the addressability data returned by FSH\_PROBEBUF is only valid until the FSD blocks. Blocking, either explicitly or implicitly allows other threads to run, and possibly invalidate a user segment. Therefore, FSH\_PROBEBUF must be reapplied after every block.

FSH\_PROBEBUF provides a convenient method for assuring that a user transfer address is valid and present in memory. Upon successful return, the user address may be treated as a far pointer and accessed up to the specified length without either blocking or faulting. This state of affairs is guaranteed up until the FSD returns or until the next block.

Once FSH\_PROBEBUF detects a protection violation, the process will be killed as soon as possible. The OS/2 kernel will kill the process once it has exited from the FSD.

On 80386 processors, FSH\_PROBEBUF will insure that all touched pages are physically present in memory, so that the FSD will not suffer an implicit block due to a page fault. However, FSH\_PROBEBUF does NOT guarantee that the pages will be physically contiguous in memory, since FSDs are not expected to do DMA.

```
int far pascal FSH_PROBEBUF (operation, pData, cbData)
unsigned short operation;
char far * pData;
unsigned short cbData;
```

Where

operation indicates whether read or write access is desired

operation == 0 indicates read access is to be checked.

operation == 1 indicates write access is to be checked.

All other values are reserved.

pData       starting address of user data

+        cbData       length of user data

+        If cbData is 0, a length of 64K is indicated.

**Returns** Error code if either the access to the data is inappropriate or the user transfer region itself is partially or completely inaccessible. 0 otherwise.

ERROR\_PROTECTION\_VIOLATION - access to the indicated memory region is illegal.

**Note** FSH\_PROBEBUF may block.

To help avoid deadlocks, FSH\_PROBEBUF should not be called while owning an OS/2 buffer.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH\_PROBEBUF where appropriate.

\* 2.1.8.12.24 FSH\_QSYSINFO - Query system information:

\* FSH\_QSYSINFO queries the system about dynamic system variables and static system variables not returned by DOSQSYSINFO.

```
* int far pascal FSH_QSYSINFO (index,pData,cbData)
* unsigned short index;
* char far * pData;
* unsigned short cbData;
```

\* Where

\* index variable to return

| index == 1 indicates maximum sector size

| index == 2 indicates process identity. The data returned will be as follows:

```
| struct {
| unsigned short PID;
| unsigned short UID;
| unsigned short PDB
| };
```

| index == 3 indicates absolute thread number for the current thread. This will be returned in an unsigned short field.

| index == 4 indicates verify on write flag for the process. This will be returned in an unsigned char (byte) field. 0 means verify is off, non-0 means it is on.

\* pData long address to data area

\* cbData length of data area

\* Returns Error code if error detected, 0 otherwise.

\* ERROR\_INVALID\_PARAMETER - invalid index.

\* ERROR\_BUFFER\_OVERFLOW - the specified buffer is too short for the returned data.

\* Note Reminder: OS/2 does not validate input parameters, so FSD should call FSH\_PROBEBUF where appropriate.

EP 0 415 346 A2

\* 2.1.8.12.25 FSH\_RELEASEBUF - Release the owned buffer:

\* FSH\_RELEASEBUF releases a buffer owned by the calling thread if any is  
\* owned.

void far pascal FSH\_RELEASEBUF ()

Returns Nothing

Note See note under FSH\_GETBUF for interactions with other buffer  
calls.

FSH\_RELEASEBUF will not block.

2.1.8.12.26 FSH\_REMOVESHARE - Remove a share entry:

FSH\_REMOVESHARE is used to remove a previously-entered name from the sharing  
set.

void far pascal FSH\_REMOVESHARE (hShare)  
unsigned long hShare;

Where

hShare share handle returned by a prior call to FSH\_ADDSHARE.

Returns None

\* Note Once a call to FSH\_REMOVESHARE has been issued, the share handle  
\* is no longer valid.

\* FSH\_REMOVESHARE may block.

\* Reminder: OS/2 does not validate input parameters, so FSD should  
\* call FSH\_PROBEBUF where appropriate.

To help avoid deadlocks, FSH\_REMOVESHARE should not be called  
while owning an OS/2 buffer.

#### 2.1.8.12.27 FSH\_SEGALLOC - Allocate a GDT or LDT segment:

\* An FSD may call FSH\_SEGALLOC to allocate a GDT or LDT selector. The  
\* selector will have read/write access.

```
int far pascal FSH_SEGALLOC (flags, cbSeg, pSel)
unsigned short flags;
unsigned long cbSeg;
unsigned short far * pSel;
```

#### Where

| flags | indicates | GDT/LDT,<br>swappable/non-swappable | protection | ring, |
|-------|-----------|-------------------------------------|------------|-------|
|-------|-----------|-------------------------------------|------------|-------|

Bit 0x0001 off indicates GDT selector returned.

Bit 0x0001 on indicates LDT selector returned.

Bit 0x0002 off indicates non-swappable memory

Bit 0x0002 on indicates swappable memory.

Bits 13 and 14 are the desired ring number.

All other bits are reserved and must be zero.

|       |                       |
|-------|-----------------------|
| cbSeg | length of the segment |
|-------|-----------------------|

|      |                                                                    |
|------|--------------------------------------------------------------------|
| pSel | far address of location where allocated selector will be<br>stored |
|------|--------------------------------------------------------------------|

Returns Error code if allocation failed, 0 otherwise.

\* ERROR\_INTERRUPT - the current thread received a signal

\* ERROR\_INVALID\_PARAMETER - reserved bits in flags are set or  
\* requested size is too large

ERROR\_NOT\_ENOUGH\_MEMORY - too much memory is allocated

#### Note

It is strongly suggested that FSDs allocate all their data at  
ring 0 for maximal protection from user programs.

GDT selectors are a scarce resource; the FSD must be prepared to

expect an error for allocation of a GDT segment. The FSD should  
limit itself to at most 10 total GDT segments. It is suggested  
that for each type of data, a large segment be allocated and  
divided into per-process records.

\* FSH\_SEGALLOC may block.

\* Reminder: OS/2 does not validate input parameters, so FSD should  
\* call FSH\_PROBEBUF where appropriate.

Care must be taken to avoid deadlocks between swappable segments  
and swapper requests.

To help avoid deadlocks, FSH\_SEGALLOC should not be called while  
owning an OS/2 buffer.

#### 2.1.8.12.28 FSH\_SEGFREE - Release a GDT or LDT segment:

An FSD may call FSH\_SEGFREE to release a segment previously allocated with FSH\_SEGALLOC, or loaded as part of the FSD image.

```
int far pascal FSH_SEGFREE (sel)
unsigned short sel;
```

##### Where

sel            selector to be freed

Returns   Error code if free failed, 0 otherwise.

ERROR\_INVALID\_ACCESS - the selector is invalid

Note        FSH\_SEGFREE may block.

\*        Reminder: OS/2 does not validate input parameters, so FSD should  
\*        call FSH\_PROBEBUF where appropriate.

#### 2.1.8.12.29 FSH\_SEGREALLOC - Change segment size:

An FSD may call FSH\_SEGREALLOC to change the size of a segment previously allocated with FSH\_SEGALLOC, or loaded as part of the FSD image. The segment may be grown or shrunk. The segment may be moved in the process. When grown, the extra space is uninitialized.

```
int far pascal FSH_SEGREALLOC (sel, cbSeg)
unsigned short sel;
unsigned long cbSeg;
```

##### Where

sel            selector of segment to be changed

cbSeg          new size to set for the segment.

Returns   Error code if free failed, 0 otherwise.

ERROR\_NOT\_ENOUGH\_MEMORY - too much memory is allocated

ERROR\_INVALID\_ACCESS - the selector is invalid

Note        FSH\_SEGREALLOC may block.

To help avoid deadlocks, FSH\_SEGREALLOC should not be called while owning an OS/2 buffer.

\*        Reminder: OS/2 does not validate input parameters, so FSD should  
\*        call FSH\_PROBEBUF where appropriate.

2.1.8.12.30 FSH\_SEMCLEAR - Clear a semaphore:

FSH\_SEMCLEAR allows an FSD to release a semaphore that was previously obtained via FSH\_SEMREQUEST.

```
int far pascal FSH_SEMCLEAR (pSem)
char far * pSem;
```

Where

pSem      handle to system semaphore or long address of ram semaphore

Returns    Error code if free failed, 0 otherwise.

ERROR\_EXCL\_SEM\_ALREADY\_OWNED - the exclusive semaphore is already owned.

ERROR\_PROTECTION\_VIOLATION - the semaphore is inaccessible.

Note       FSH\_SEMCLEAR may block.

To help avoid deadlocks, FSH\_SEMCLEAR should not be called while owning an OS/2 buffer.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH\_PROBEBUF where appropriate.

\* 2.1.8.12.31 FSH\_SEMREQUEST - Request a semaphore:

\* FSH\_SEMREQUEST allows an FSD to obtain exclusive access to a semaphore.

```
int far pascal FSH_SEMREQUEST (pSem, cmsTimeout)
char far * pSem;
unsigned long cmsTimeout;
```

Where

pSem      handle to system semaphore or long address of ram semaphore

cmsTimeout    number of milliseconds to wait

Returns    Error code if failed, 0 otherwise.

ERROR\_INTERRUPT - the current thread received a signal

ERROR\_SEM\_TIMEOUT - the timeout expired without gaining access to the semaphore.

ERROR\_SEM\_OWNER\_DIED - the owner of the semaphore died.

ERROR\_TOO\_MANY\_SEM\_REQUESTS - there are too many semaphore requests in progress.

ERROR\_PROTECTION\_VIOLATION - the semaphore is inaccessible.

Note       The distinguished timeout value of 0xFFFFFFFF indicates an infinite timeout.

\*        The caller may receive access to the semaphore after the timeout period has expired without receiving an ERROR\_SEM\_TIMEOUT. Therefore, semaphore timeout values should not be used for exact timing and sequencing.

\*        FSH\_SEMREQUEST may block.

\*        To help avoid deadlocks, FSH\_SEMREQUEST should not be called while owning an OS/2 buffer.

\*        Reminder: OS/2 does not validate input parameters, so FSD should call FSH\_PROBEBUF where appropriate.

#### 2.1.8.12.32 FSH\_SEMSET - Set a semaphore:

FSH\_SEMSET allows an FSD to set a semaphore unconditionally.

```
int far pascal FSH_SEMSET (pSem)
char far * pSem;
```

##### Where

pSem      handle to system semaphore or long address of ram  
            semaphore

Returns   Error code if invalid semaphore, 0 otherwise.

ERROR\_EXCL\_SEM\_ALREADY\_OWNED - the exclusive semaphore is  
already owned.

ERROR\_TOO\_MANY\_SEM\_REQUESTS - there are too many semaphore  
requests in progress.

ERROR\_PROTECTION\_VIOLATION - the semaphore is inaccessible.

Note      FSH\_SEMSET may block.

To help avoid deadlocks, FSH\_SEMSET should not be called while  
owning an OS/2 buffer.

Reminder: OS/2 does not validate input parameters, so FSD should  
call FSH\_PROBEBUF where appropriate.

#### 2.1.8.12.33 FSH\_SEMSETWAIT - Set a semaphore and wait for clear:

FSH\_SEMSETWAIT allows an FSD to wait for an event. The event is signalled  
by a call to FSH\_SEMCLEAR.

```
int far pascal FSH_SEMSETWAIT (pSem, cmsTimeout)
char far * pSem;
unsigned long cmsTimeout;
```

##### Where

pSem      handle to system semaphore or long address of ram  
            semaphore

cmsTimeout   number of milliseconds to wait

Returns   Error code if timeout occurred or invalid semaphore, 0 otherwise.

ERROR\_SEM\_TIMEOUT - the timeout expired without gaining access  
to the semaphore.

ERROR\_EXCL\_SEM\_ALREADY\_OWNED - the exclusive semaphore is  
already owned.

ERROR\_INTERRUPT - the current thread received a signal

ERROR\_PROTECTION\_VIOLATION - the semaphore is inaccessible.

##### \* Note

The caller may return after the timeout period has expired without  
receiving an ERROR\_SEM\_TIMEOUT. Therefore, semaphore timeout  
values should not be used for exact timing and sequencing.

FSH\_SEMSETWAIT may block.

To help avoid deadlocks, FSH\_SEMSETWAIT should not be called while  
owning an OS/2 buffer.

Reminder: OS/2 does not validate input parameters, so FSD should  
call FSH\_PROBEBUF where appropriate.

#### 2.1.8.12.34 FSH\_SEMWAIT - Wait for clear:

FSH\_SEMWAIT allows an FSD to wait for an event. The event is signalled by a call to FSH\_SEMCLEAR.

```
int far pascal FSH_SEMWAIT (pSem, cmsTimeout)
char far * pSem;
unsigned long cmsTimeout;
```

##### Where

pSem handle to system semaphore or long address of ram semaphore

cmsTimeout number of milliseconds to wait

Returns Error code if free failed, 0 otherwise.

ERROR\_SEM\_TIMEOUT - the timeout expired without gaining access to the semaphore.

ERROR\_INTERRUPT - the current thread received a signal

ERROR\_PROTECTION\_VIOLATION - the semaphore is inaccessible.

Note The caller may return after the timeout period has expired without receiving an ERROR\_SEM\_TIMEOUT. Therefore, semaphore timeout values should not be used for exact timing and sequencing.

FSH\_SEMWAIT may block.

To help avoid deadlocks, FSH\_SEMWAIT should not be called while owning an OS/2 buffer.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH\_PROBEBUF where appropriate.

#### 2.1.8.12.35 FSH\_STORECHAR - Store a character in a string:

\* Provides mechanism for storing a character into a string taking into account  
\* DBCS considerations.

```
* void far pascal FSH_STORECHAR (chDBCS, ppStr)
unsigned short chDBCS;
char far * far * ppStr;
```

##### Where

chDBCS character to be stored. This may be either a single byte character or a double byte character with the first byte occupying the low-order position.

ppStr pointer to character pointer where character will be stored. This pointer is updated upon return.

\* Note The FSD is responsible for verifying the string pointer and checking for segment boundaries.

\* Reminder: OS/2 does not validate input parameters, so FSD should call FSH\_PROBEBUF where appropriate.



2.1.8.12.36 FSH\_UPPERCASE - Uppercase asciiz string:

FSH\_UPPERCASE is used to uppercase an asciiz string.

```
int far pascal FSH_UPPERCASE (szPathName, cbPathBuf, pPathBuf)
char far * szPathName;
unsigned short cbPathBuf;
char far * pPathBuf;
```

Where

szPathName pointer to asciiz pathname to be uppercased.

cbPathBuf length of pathname buffer.

pPathBuf pointer to buffer to copy uppercased path into.

Returns Error code if error detected, 0 otherwise.

ERROR\_BUFFER\_OVERFLOW - uppercased pathname is too long to fit in buffer.

Note This routine processes DBCS characters properly.

The FSD is responsible for verifying the string pointers and checking for segment boundaries.

szPathName and pPathBuf may point to the same place in memory.

FSH\_UPPERCASE should be called for names passed into the FSD in raw data packets which are not passed to FSH\_CANONICALIZE and should be uppercased, i.e. extended attribute names.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH\_PROBEBUF where appropriate.

2.1.8.12.37 FSH\_WILDMATCH - Match using OS/2 wildcards:

Provides mechanism for using OS/2 wildcards semantics to form a match between an input string and a pattern, taking into account DBCS considerations.

Wildcards provide a general mechanism for pattern matching file names. There are two distinguished characters that are relevant to this matching. The '?' character matches one character (not byte) except at a '.' or at the end of a string, where it matches zero characters. The '\*' matches zero or more characters (not bytes) with no implied boundaries except the end-of-string. See discussion of meta characters.

For example, "a\*b" matches "ab" and "aCCCCCCCCCb" while "a?b" matches "aCb" but does not match "aCCCCCCCCCb".

FSH\_WILDMATCH is case-sensitive.

```
int far pascal FSH_WILDMATCH (pPat, pStr)
char far * pPat;
char far * pStr;
```

Where

pPat pointer to asciiz pattern string. Wildcards are present and are interpreted as described above.

pStr pointer to test string.

Returns Error code if match failed, 0 otherwise.

ERROR\_NO\_META\_MATCH - the wildcard match failed.

Note Reminder: OS/2 does not validate input parameters, so FSD should call FSH\_PROBEBUF where appropriate.

#### 2.1.8.12.38 FSH\_YIELD - Yield CPU to higher priority threads:

Provides mechanism for relinquishing the processor to higher-priority threads. FSDs run under the 2ms dispatch latency imposed on the OS/2 kernel, meaning that no more than 2ms can be spent in an FSD without an explicit block or yield. FSH\_YIELD will test to see if another thread is runnable at the current thread's priority or at a higher priority. If one exists, that thread will be given a chance to run.

```
void far pascal FSH_YIELD ()
```

Returns None.

#### 2.1.8.13 Overview

This package describes the changes to the boot architecture and extensions to the Installable File System (IFS) mechanism to enable booting from an File System Driver (FSD) managed volume (referred to as Bootable IFS. If the volume is on a remote system, it is referred to as Remote IPL.)

This design centers around a new component referred to as the mini-FSD. The mini-FSD is similar to the FSD defined in the IFS spec. However, it has additional requirements placed on it for reading the base device drivers. These requirements are fully defined in the interfaces section.

To satisfy its I/O requests, the mini-FSD may call the disk device driver imbedded in OS2KRNL (the bootable IFS case) or it may contain its own imbedded device driver (the remote IPL case).

Along with the mini-FSD, a new utility referred to as the IFS SYS utility is required to initialize an FSD managed volume with whatever is required to satisfy the requirements of the mini-FSD and this spec.

The IFS mechanism must be expanded slightly to include some additional calls which the mini-FSD may need while it is linked into the IFS chain.

#### 2.1.8.14 Operational Description

In order to establish a frame of reference, the present boot procedure (which uses the FAT file system) will be described first.

#### 2.1.8.15 FAT Boot Procedure

The following figure represents the various major stages during the FAT boot procedure.

```
|-----|-----|-----|-----|-----|-----> time
 POST IBMBOOT OS2LDR stage1 stage2 stage3
 OS2KRNL->
```

Power applied to the machine, or pressing CTRL-ALT-DEL, causes control to get transferred to the power-on-self-test (POST) code. This code, among other things, initializes the interrupt vectors to get to the BIOS routines. It then scans the IO adapters looking for and linking in any code which exists on them. It then executes an interrupt 19h (INT 19) which causes control to be transferred to the reboot code.

The BIOS' interrupt 19h handler will read the boot sector from disk into memory at 7C00h and transfer control to it.

The boot sector contains a data structure (BPB) which describes the disk's parameters. It also contains code to read in the root directory, scan for OS2LDR, and read it into memory. The loader is very simple and requires that OS2LDR be located in contiguous sectors. It uses the BIOS' interrupt 13h handler to do reads from the disk. Control is then transferred to OS2LDR along with a pointer to the BPB and the drive number required by interrupt 13h.

OS2LDR contains the OS/2 loader. It relocates itself to the top of low memory, then scans the root directory for OS2KRNL and reads it into memory. No restrictions are imposed on OS2KRNL. Again, it is loaded via BIOS's interrupt 13h handler. After the required fixups are applied, control is transferred to OS2KRNL along with a pointer to the BPB and the drive number.

OS2KRNL contains the OS/2 kernel and initialization code. It switches to protected mode, relocates parts of itself to high memory, then scans the root directory for and reads in the base device drivers (stage 1). Once again, BIOS's interrupt 13h is used to read the disk, but mode switching must be done.

OS2KRNL then switches to protection ring 3 and loads some of the required dynamic link libraries (stage 2) followed by the device drivers and file system drivers specified in CONFIG.SYS (stage 3). This is done with standard DOS calls and therefore goes thru the regular file system and device drivers.

#### 2.1.8.16 Non-FAT Boot Procedure

The following figure represents the various major stages during the non-FAT boot procedure.

```
|-----|-----|-----|-----|-----|-----> time
| POST "black OS2LDR stagel stage2 stage3
| box" OS2KRNL->
```

In order to enable both bootable IFS and remote IPL, the boot architecture is modified to assume nothing about the sequence of events from the time when POST executes the interrupt 19h to the time OS2LDR receives control (ie. the "black box" is running). However, it does assume that a memory image of the files OS2KRNL and the mini-FSD are in memory when OS2LDR is given control. The memory map must be as follows:

```
| +-----+ Top of Low Memory
| | reserved |
| +-----+
| | OS2LDR |
| +-----+ paragraph boundary
| | OS2KRNL |
| +-----+ 10000h
| | mini-FSD |
| +-----+ 7C0h
| | reserved |
| +-----+ 0
```

The mini-FSD must be loaded in the range 7C0h to 0FFFFh. OS2KRNL must be loaded at 10000h. OS2LDR must be loaded on the next paragraph boundary above OS2KRNL.

When OS2LDR receives control, it must also receive a pointer to the file images for OS2KRNL and the mini-FSD. If the mini-FSD intends to use MFSH\_DOVOLIO to do disk reads, it must also be passed the pointer to the BPB and the drive number.

#### 2.0 Functional Characteristics

351

OS2LDR will relocate the mini-FSD to high memory (above the 1-Meg boundary), then continue on with the same initialization before transferring control to OS2KRNL.

When OS2KRNL receives control, it will go through the same initialization as before (stage 1) with a couple of exceptions. The module loader will be called to "load" the mini-FSD from its memory image stored by OS2LDR in high memory. Also, the mini-FSD will be called to read the base device drivers (one at a time) via the stage 1 interfaces.

Before any of the dynamic link libraries are loaded, the mini-FSD will be linked into the IFS chain (actually, it will be the only link in the chain.) and asked to initialize via FS\_INIT. The FS\_INIT call marks the transition from stage 1 to stage 2.

The dynamic link libraries are then loaded using the stage 2 interfaces, followed by the device drivers and file system drivers.

The mini-FSD is required to support only a small number of the functions of an FSD. Therefore, the first FSD loaded must be the replacement for the mini-FSD.

After the replacement FSD has been loaded, it will be called at FS\_INIT to initialize itself and take whatever action it needs to effect a smooth transition from the mini-FSD to the FSD. The FSD will then replace the mini-FSD in the IFS chain, as well as in any kernel data structures which keep a handle to the FSD (ie. SFT, VPB).

From this point on, the system continues normally.

#### 2.1.8.17 Interfaces

The mini-FSD is built as a dynamic link library. Supplied functions are exported by making the function names public. Helper functions are imported by declaring the helper names external:far. It is required only to support reading files and will be called only in protect mode. The mini-FSD may NOT make dynamic link system calls at init time.

Due to the special state of the system as it boots, the programming model for the mini-FSD during the stage 1 time frame is somewhat different than the model for stage 2. This difference necessitates 2 different interfaces between OS/2 and the mini-FSD.

During stage 1, all calls to the mini-FSD will be to the MFS\_xxxx functions. Only the MFSH\_xxxx helper functions are available. These are the interfaces which are addressed in this document. Many of these interfaces parallel the

#### 2.0 Functional Characteristics

352

00181

EP 0 415 346 A2

interfaces defined in the IFS spec while others are unique to the mini-FSD.

During stage 2, the mini-FSD is treated as a normal FSD. Calls will be to the FS\_xxxx functions and all FSH\_xxxx helper functions are available. Since these interfaces are covered in the IFS spec, they will not be repeated here.

During stage 3, the mini-FSD is given a chance to release resources (via a call to MFS\_TERM) and before being terminated.

Transition from stage 1 to stage 2 is marked by calling the FS\_INIT function in the mini-FSD. Transition from stage 2 to stage 3 is marked by calling FS\_INIT in the FSD.

The following figure shows the functions called during a typical boot sequence:

| mini-FSD       | FSD             |
|----------------|-----------------|
| stage 1        | stage 2         |
| MFS_INIT       |                 |
| MFS_OPEN       |                 |
| MFS_READ       |                 |
| MFS_CHGFILEPTR |                 |
| MFS_CLOSE      |                 |
|                | FS_INIT         |
|                | FS_MOUNT/ATTACH |
|                | FS_OPEN         |
|                | FS_READ         |
|                | FS_CHGFILEPTR   |
|                | FS_INIT         |
|                | MFS_TERM        |
|                | FS_READ         |

Note that no files are open at the transition from stage 1 to stage 2. Also, only a single file at a time will be open during stage 1.

Note that files and volumes will be open during the transition from stage 2 to stage 3 (the mini-FSD to the FSD). The FSD must do whatever is necessary for it to "inherit" them. The FSD will NOT receive mounts/attaches or opens for volumes and files which were mounted/attached and opened by the mini-FSD. Also, multiple files may be open simultaneously during stages 2 and 3.

A special set of helper functions have been made available to the mini-FSD to support an imbedded device driver. This might be required for situations such as remote IPL where the boot volume is not readable via DOVOLIO. These special helper functions (referred to as imbedded device driver helpers) are available during all stages of the mini-FSD's life.

Since the mini-FSD is a new component being added to the boot sequence, a

new interface to OS2LDR is required.

The name and attributes of the mini-FSD must match EXACTLY the name and attributes of the replacement FSD.

Calling conventions for all interfaces except OS2LDR are as specified in the IFS spec.

#### 2.1.8.18 Black Box to Mini-FSD Interface

A data structure (it will be referred to as "BootData" from here on) may be passed from the black box to the mini-FSD by including a pointer to it in the resource table (described in the OS2LDR interface below.) It must initially reside in the memory reserved for the mini-FSD's use ( 7C0h -OFFFh). OS2LDR will relocate it to high memory in a segment all its own. A pointer to BootData will be available to the mini-FSD via the MFS\_INIT interface.

#### 2.1.8.19 OS2LDR Interface

When initially transferring control to OS2LDR, the following interface is defined:

|       |                                                                                                         |
|-------|---------------------------------------------------------------------------------------------------------|
| DH    | boot mode flags (see below).                                                                            |
| BX    | logical sector number of first sector of cluster number 2 (ignored if DH[2] = 1).                       |
| DL    | drive ID (0 for drive A, 80 for drive C) for boot volume (ignored if DH[0] = 1).                        |
| DS:SI | pointer to the boot media's BPB (ignored if DH[0] = 1).                                                 |
| ES:DI | pointer to the resource table (if DH[2] = 1) or pointer to a copy of the root directory (if DH[2] = 0). |

The boot mode flags are defined as follows:

|          |                                                                                                                                                            |
|----------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| bits 7-3 | reserved; must be zero.                                                                                                                                    |
| bit 2    | Flag to indicate the presence (1) or absence (0) of a mini-FSD.                                                                                            |
| bit 1    | Flag to indicate if the boot volume is local (0) or remote (1). This flag is used to inform OS/2 that a remote IPL is in progress. (Ignored if DH[2] = 0.) |
| bit 0    | Flag to indicate if the mini-FSD will use OS2LDR's disk device                                                                                             |

driver (0) or its own imbedded device driver (1). (Ignored if  
DH[2] = 0.)

The format of the BPB is defined in the IFS spec and is referred to as  
"Extended\_BPB".

The resource table is defined as follows:

|       |                                                            |
|-------|------------------------------------------------------------|
| WORD  | number of entries in this table (3 or 4).                  |
| WORD  | paragraph number where the OS2LDR file image was loaded.   |
| DWORD | length of the OS2LDR file image (in bytes).                |
| WORD  | paragraph number where the OS2KRNL file image was loaded.  |
| DWORD | length of the OS2KRNL file image (in bytes).               |
| WORD  | paragraph number where the mini-FSD file image was loaded. |
| DWORD | length of the mini-FSD file image (in bytes).              |
| WORD  | paragraph number of BootData.                              |
| DWORD | length of BootData (in bytes).                             |

#### 2.1.8.20 Stage 1 Interfaces

The following functions must be made available by the mini-FSD. These  
functions will be called only during stage 1.

- MFS\_CHGFILEPTR
- MFS\_CLOSE
- MFS\_INIT
- MFS\_OPEN
- MFS\_READ
- MFS\_TERM

The following helper functions are available to the mini-FSD. These  
functions may be called only during stage 1.

- MFSH\_DOVOLIO
- MFSH\_INTERR
- MFSH\_SECALLLOC
- MFSH\_SEGFREE
- MFSH\_SEGREALLOC

#### 2.1.8.20.1 MFS\_CHGFILEPTR:

Purpose Move the file's logical read position pointer.

Calling Sequence

```
PUSH offset
PUSH type
CALL far ptr MFS_CHGFILEPTR
```

Where

offset      DWORD (signed) to be added to the current file position  
to form the new position within the file.

type        WORD indicating the base of the seek operation.

0 indicates seek relative to the beginning of the file.

1 indicates seek relative to the current position within  
the file.

2 indicates seek relative to the end of the file.

Returns Error code if function failed, 0 otherwise.

Remarks None.

#### 2.1.8.20.2 MFS\_CLOSE:

Purpose Close the file.

Calling Sequence

```
CALL far ptr MFS_CLOSE
```

Returns Error code if function failed, 0 otherwise.

Remarks None.

#### 2.1.8.20.3 MFS\_INIT:

Purpose Inform the mini-FSD that it should prepare itself for use.

Calling Sequence

```
PUSH# BootData
PUSH# ReservedDrives
PUSH# VectorIPL
PUSH# BPB
PUSH# pMiniFSD
```

PUSH@ DumpRoutine  
CALL far ptr MFS\_INIT

Where

BootData Data passed from black box to mini-FSD (null if not passed).

ReservedDrives BYTE which may be filled in by the mini-FSD with the number of drive letters (beginning with "C") to skip over before assigning drive letters to local fixed disk drives (ignored if not remote IPL). The system will attach the reserved drives to the mini-FSD via a call to FS\_ATTACH just after the call to FS\_INIT.

VectorIPL DWORD which may be filled in by the mini-FSD with a pointer to a data structure which will be available to installable device drivers via the standard device helper function "GetDOSVar" (variable number 12). The first eight bytes of the structure MUST be a signature which would allow unique identification of the data by co-operating device drivers (ie. "IBMPCNET").

BPB BPB data structure (see OS2LDR interface).

pMiniFSD DWORD which is filled in by the mini-FSD with data to be passed on to the FSD.

DumpRoutine DWORD which is filled in by the mini-FSD with the address of an alternative stand-alone dump procedure.

Returns Error code if function failed, 0 otherwise.

Remarks The mini-FSD should fill in the data pointed to by pMiniFSD with any 32-bit value it wishes to pass on to the FSD (see FS\_INIT). OS/2 makes no assumptions about the type of data passed. Typically, this will be a pointer to important data structures within the mini-FSD which the FSD needs to know about.

OS/2 will not free the segment containing BootData. It should be freed by the mini-FSD if appropriate.

The DumpProcedure is a routine provided by the mini-FSD which replaces the diskette based OS/2 stand-alone dump procedure. This routine is given control after the OS/2 kernel receives a stand-alone dump request. The OS/2 kernel places the machine in a stable, real mode state in which most interrupt vectors contain their original power-up value. If this address is left at zero, the OS/2 kernel will attempt to initiate a storage dump to diskette, if a diskette drive exists. The provided routine must handle the dumping of storage to an acceptable media.

2.1.8.20.4 MFS\_OPEN:

Purpose Open the specified file.

Calling Sequence

PUSH@ Name  
PUSH@ Size  
CALL far ptr MFS\_OPEN

Where

Name ASCIIZ name of the file to be opened. It may include a path but will not include a drive.

Size DWORD which is filled in by the mini-FSD with the size of the file in bytes.

Returns Error code if function failed, 0 otherwise.

Remarks Only one file at a time will be open via this interface. The drive will always be the boot drive.

The current file position is set to the beginning of the file.

2.1.8.20.5 MFS\_READ:

Purpose Read the specified number of bytes from the file to a data area.

Calling Sequence

PUSH@ Data  
PUSH@ Length  
CALL far ptr MFS\_READ

Where

Data data area. The data area is guaranteed to be below the 1-Meg boundary.

Length WORD which specifies the number of bytes to be read. On return, it has been filled in by the mini-FSD with the number of bytes successfully read.

Returns Error code if function failed, 0 otherwise.

Remarks The current file position is advanced by the number of bytes read.

#### 2.1.8.20.6 MFSH\_DOVOLIO:

**Purpose** Read the specified sectors from the boot volume into a data area.

**Calling Sequence**

```
PUSH@ Data
PUSH@ cSec
PUSH iSec
CALL far ptr MFSH_DOVOLIO
```

**Where**

**Data** data area. The data area must be below the 1-Meg boundary.

**cSec** WORD which specifies the number of sectors to be read. On return, it has been filled in by the helper with the number of sectors successfully read.

**iSec** DWORD which is the sector number of the first sector.

**Returns** Error code if function failed, 0 otherwise.

**ERROR\_PROTECTION\_VIOLATION** - the supplied address or length is invalid.

**Remarks** The only media which can be read via this interface is the boot volume. The machine's interrupt 13h BIOS function is used to actually do the disk reads. The data area will be locked and unlocked by this helper. Soft errors are retried automatically. Hard errors are reported to the user via a message and the system is stopped.

#### 2.1.8.20.7 MFSH\_INTERR:

**Purpose** Display a message on the screen and stops the system. This function should be used when an inconsistency is detected within the mini-FSD.

**Calling Sequence**

```
PUSH@ Msg
PUSH cbMsg
CALL far ptr MFSH_INTERR
```

**Where**

**Msg** message text.

**cbMsg** WORD which is the length of the message text.

**Returns** Does not return.

**Remarks** See the notes under FSH\_INTERR in the IFS spec.

#### 2.1.8.20.8 MFSH\_SEGALLOC:

**Purpose** Allocate a segment of memory.

**Calling Sequence**

```
PUSH Flag
PUSH cbSeg
PUSH@ Sel
CALL far ptr MFSH_SEGALLOC
```

**Where**

**Flag** WORD which is one (1) if the memory must be below the 1-Meg boundary or zero (0) if it's location doesn't matter.

**cbSeg** DWORD which is the length of the segment.

**Sel** WORD which is filled in by the helper with the selector.

**Returns** Error code if function failed, 0 otherwise.

**ERROR\_NOT\_ENOUGH\_MEMORY** - too much memory is already allocated.

**ERROR\_PROTECTION\_VIOLATION** - the supplied address is invalid.

**Remarks** The memory allocated will have the following attributes: GDT, ring 0, non-swappable.

Memory not allocated specifically below the 1-Meg boundary may be "given" to the FSD by passing the selector(s) via pMiniFSD (see MFS\_INIT and FS\_INIT).

#### 2.1.8.20.9 MFSH\_SEGFREE:

**Purpose** Release a segment previously allocated with MFSH\_SEGALLOC, or loaded as part of the mini-FSD image.

**Calling Sequence**

```
PUSH Sel
CALL far ptr MFSH_SEGFREE
```

Where

Sel WORD which is the selector to be freed.

Returns Error code if function failed, 0 otherwise.

ERROR\_PROTECTION\_VIOLATION - the supplied selector is invalid.

Remarks None

2.1.8.20.10 MFSH\_SEGREALLOC:

Purpose Change the size of a segment previously allocated with MFSH\_SEGALLOC, or loaded as part of the mini-FSD image.

PUSH Sel

PUSH cbSeg

CALL far ptr MFSH\_SEGREALLOC

Where

Sel WORD which is the selector.

cbSeg DWORD which is the length of the segment.

Returns Error code if function failed, 0 otherwise.

ERROR\_NOT\_ENOUGH\_MEMORY - too much memory is already allocated.

ERROR\_PROTECTION\_VIOLATION - the supplied selector is invalid.

Remarks The segment may be grown or shrunk. When grown, the extra space is uninitialized. The segment may be moved in the process.

2.1.8.21 Stage 2 Interfaces

The intent of stage 2 is to use the mini-FSD as an FSD. Therefore, all the guidelines and interfaces specified in the IFS spec apply with the following exceptions.

The following IFS functions must be fully supported by the mini-FSD:

- \* FS\_ATTACH (remote mini-FSD only)
- \* FS\_ATTRIBUTE
- \* FS\_CHGFILEPTR

- \* FS\_CLOSE
- \* FS\_COMMIT
- \* FS\_INIT
- \* FS\_IOCTL
- \* FS\_MOUNT (local mini-FSD only)
- \* FS\_NAME
- \* FS\_OPENCREATE (existing file only)
- \* FS\_PROCESSNAME
- \* FS\_READ

Note that since the mini-FSD is only required to support reading, FS\_OPENCREATE need only support opening an existing file (not the create or replace options).

None of the other functions required by the IFS spec are required of the mini-FSD but must be defined and should return the ERROR\_UNSUPPORTED\_FUNCTION return code.

The full complement of helper functions specified in the IFS spec is available to the mini-FSD. However, the mini-FSD may NOT use any other dynamic link calls.

2.1.8.22 Stage 3 Interfaces

The intent of stage 3 is to throw away the mini-FSD and use only the FSD.

The following IFS functions must be supported by the mini-FSD:

- \* MFS\_TERM

2.1.8.22.1 MFS\_TERM:

Purpose Inform the mini-FSD that it should prepare itself for termination.

Calling Sequence

CALL far ptr MFS\_TERM

Returns Error code if function failed, 0 otherwise.

Remarks The system will NOT free any memory explicitly allocated by the mini-FSD via MFSH\_SEGALLOC or FSH\_SEGALLOC; it must be explicitly freed by the mini-FSD. (Memory allocated by the mini-FSD and "given" to the FSD need not be freed.) The system will free all of the segments loaded as part of the mini-FSD image immediately



after this call.

#### 2.1.8.23 Imbedded Device Driver Helpers

The following helper functions are available to the mini-FSD and may be called during stage 1, 2 or 3. These helpers are counterparts for some of the device help functions and are intended for use by a device driver imbedded within the mini-FSD.

- \* MFSH\_CALLRM
- \* MFSH\_LOCK
- \* MFSH\_PHYSTOVIRT
- \* MFSH\_UNLOCK
- \* MFSH\_UNPHYSTOVIRT
- \* MFSH\_VIRTTOPHYS

##### 2.1.8.23.1 MFSH\_CALLRM:

**Purpose** Put the machine into real mode, call the specified routine, put the machine back into protect mode and return.

**Calling Sequence**  
PUSH@ Proc  
CALL far ptr MFSH\_CALLRM

**Where**  
  
Proc        DWORD    which is the PHYSICAL address of the routine to call.

**Returns** Undefined.

**Remarks** Only registers DS and SI will be preserved between the caller and the target routine. The selectors in the segment registers will be converted to segments before calling the target routine. Arguments may not be passed on the stack since a stack switch may occur.

This helper allows the mini-FSD to access the ROM BIOS' functions which typically run in real mode only. Great care must be taken in using this function since selectors used through-out the system are meaningless in real mode. While in real mode, no calls to any helpers may be made.

##### 2.1.8.23.2 MFSH\_LOCK:

**Purpose** Lock a segment in place in physical memory.

**Calling Sequence**  
PUSH Sel  
PUSH@ Handle  
CALL far ptr MFSH\_LOCK

**Where**  
  
Sel        WORD    which is the selector to be locked.  
  
Handle     DWORD    which is filled in by the helper with the lock handle.

**Returns** Error code if function failed, 0 otherwise.

ERROR\_PROTECTION\_VIOLATION - the supplied address or selector is invalid.

**Remarks** This helper is analogous to the device driver LOCK helper with the following attributes: short term, block until locked.

##### 2.1.8.23.3 MFSH\_PHYSTOVIRT:

**Purpose** Allocate a selector and bind a physical address and length to it.

**Calling Sequence**  
PUSH Addr  
PUSH Len  
PUSH@ Sel  
CALL far ptr MFSH\_PHYSTOVIRT

**Where**  
  
Addr        DWORD    which is the physical address.  
  
Len        WORD    which is the length of the segment.  
  
Sel        WORD    which is filled in by the helper with the selector.

**Returns** Error code if function failed, 0 otherwise.

ERROR\_NOT\_ENOUGH\_MEMORY - too many selectors already allocated.

ERROR\_PROTECTION\_VIOLATION - the supplied address is invalid.

EP 0 415 346 A2

Remarks This helper is somewhat analogous to the device driver PHYSTOVIRT helper. A call to PHYSTOVIRT must be paired with a call to UNPHYSTOVIRT to release the selector.

#### 2.1.8.23.4 MFSH\_UNLOCK:

Purpose Unlock a segment which was previously locked.

##### Calling Sequence

```
PUSH Handle
CALL far ptr MFSH_UNLOCK
```

##### Where

Handle DWORD which contains the handle returned from MFSH\_LOCK.

Returns Error code if function failed, 0 otherwise.

ERROR\_PROTECTION\_VIOLATION - the supplied address is invalid.

Remarks This helper is analogous to the device driver UNLOCK helper.

#### 2.1.8.23.5 MFSH\_UNPHYSTOVIRT:

Purpose Release the selector allocated by MFSH\_PHYSTOVIRT.

##### Calling Sequence

```
PUSH Sel
CALL far ptr MFSH_UNPHYSTOVIRT
```

##### Where

Sel WORD which is the selector.

Returns Error code if function failed, 0 otherwise.

ERROR\_PROTECTION\_VIOLATION - the supplied selector is invalid.

Remarks This helper is somewhat analogous to the device driver UNPHYSTOVIRT helper. A call to PHYSTOVIRT must be paired with a call to UNPHYSTOVIRT to release the selector.

#### 2.1.8.23.6 MFSH\_VIRT2PHYS:

Purpose Translate a virtual address to a physical address.

##### Calling Sequence

```
PUSH VirtAddr
PUSH PhysAddr
CALL far ptr MFSH_VIRT2PHYS
```

##### Where

VirtAddr DWORD which is the virtual address to be translated.

PhysAddr DWORD which is filled in by the helper with the physical address equivalent of the virtual address.

Returns Error code if function failed, 0 otherwise.

ERROR\_PROTECTION\_VIOLATION - the supplied address is invalid.

Remarks This helper is analogous to the device driver VIRT2PHYS helper.

#### 2.1.8.24 Special Considerations

#### 2.1.8.25 Constraints

Since the mini-FSD's file image must fit between 7C0h and 0FFFFh, the size of the mini-FSD file image may not exceed approximately 62k. Further, the memory requirements of the mini-FSD may not exceed 64k. This is an arbitrary constraint, but is required to ensure bootability under various conditions.

#### 2.1.8.26 Limitations

The mini-FSD is only required to support reading of a file. Therefore, any call to DosWrite (or other non-supported functions) which becomes redirected to the mini-FSD may be rejected. For this reason, it is required that the IFS= statement which loads the FSD which will replace the mini-FSD be the first IFS= statement in CONFIG.SYS. Further, only DEVICE= statements which load device drivers required by that FSD should appear before the first IFS= statement.

If the mini-FSD needs to switch to real mode, it must use the MFSH\_CALLRM function. This is required to keep OS/2 informed of the mode switching.

This design assumes that OS2KRNL will not grow without bounds. This design will work as long as the size of the OS2LDR, OS2KRNL pair remains below approximately 448k (assuming only 512k of low memory available in the machine. This is required by our objectives statement today, but may change in the future. If the objectives are changed to require 640k of low memory, this boundary moves up to around 576k.) This is also the reason OS2LDR must be located on the next paragraph boundary above OS2KRNL.

#### 2.1.8.27 Dependencies

Each FSD which is bootable is required to provide their own "black box" to load OS2LDR, OS2KRNL and the mini-FSD into memory before OS2LDR is given control.

Additionally, these FSDs are required to provide a fourth function in their utility dynamic link library (see the IFS spec) to support the OS/2 SYSINSTX utility. This function must do whatever is required to make the partition bootable. At the very least, it must install a boot sector. It will probably also need to install the "black box", mini-FSD, OS2LDR and OS2KRNL.

Index

+---+  
| A |  
+---+

ANSI Support for Video and Keyboard 53  
ASCII2 strings 54  
asynchronous read from a file 197  
asynchronous write to a file 223  
attach a pseudo-character device name to an FSD 153  
attach a volume to a remote file system 153  
attach drive 233  
attached remote file system, query information from a 182  
attributes, extended 101  
attributes, file 180, 112  
AUTOFAIL 84

+---+  
| B |  
+---+

BPB 66  
buffer reset 113, 244, 270  
BUFFERS 84

+---+  
| C |  
+---+

canonicalization 293  
change current directory 114  
Change directory path 236  
change file (EOD) size 162, 283  
change file read  
    write pointer 116  
close a file handle 118, 242  
close find handle 139  
close find-notify handle 148, 265  
close, find 256  
commands, configuration 83  
commit a file handle 244  
CONFIG.SYS 83  
configuration commands 83  
configuration, AUTOFAIL 84  
configuration, BUFFERS 84  
configuration, COUNTRY 85  
configuration, DEVICE 85  
configuration, protect mode 83  
configuration, real mode 83

Index

configuration, system 83  
Connect named pipe 32  
control, file system 155, 271  
cooked mode 53  
copy a file 246  
copy file 119  
COUNTRY 85  
create a file 288  
create a named pipe 34  
current directory, change 114  
current directory, query 171

+---+  
| D |  
+---+

DBCS, decrement character pointer 331  
DBCS, load character from string 329  
DBCS, match string 348  
DBCS, store character 346  
delete a file 123, 248  
detach a pseudo-character device name from an FSD 153  
detach a volume from a remote file system 153  
DEVICE 85  
device control 58, 125, 276  
device handle 53  
device name 57  
device name, character 182  
device name, pseudo 182  
devioctl 311, 316  
direct read named pipe 46  
direct write named pipe 47  
directory information, enumerate 130  
directory information, query 189  
directory information, set 216  
directory path, Change 236  
directory path, Verify 236  
directory read close 256  
directory search close 256  
directory, change 114  
Disconnect named pipe 33  
disk drive, select 205  
disk, query current 173  
dispatch latency 349  
Dos Set Code Page 71, 73  
DosBeep 58  
DosBufReset 113  
DosCallNmPipe 31  
DosCaseMap 77

Index

DosChDir 114  
DosChgFilePtr 116  
DosClose 118  
DosConnectNmPipe 32  
DosCopy 119  
DosDelete 123  
DosDevIOctl 58, 125  
DosDevIOctl2 125  
DosDisconnectNmPipe 33  
DosDupHandle 127  
DosEditName 129  
DosEnumAttribute 130  
DosFileIO 133  
DosFileLocks 137  
DosFindClose 139  
DosFindFirst 140  
DosFindFirst2 140  
DosFindNext 146  
DosFindNotifyClose 148  
DosFindNotifyFirst 149  
DosFlagProcess 25  
DosFsAttach 153  
DosFsCtl 155  
DosGetCollate 80  
DosGetCp 74  
DosGetCtryInfo 75  
DosGetDBCSEv 78  
DosMakeNmPipe 34  
DosMakePipe 28  
DosMkDir 158  
DosMkDir2 158  
DosMove 160  
DosNewSize 162  
DosOpen 163  
DosOpen2 163  
DosPeekPipe 40  
DosQCurDir 171  
DosQCurDisk 173  
DosQFHandState 174  
DosQFileInfo 177  
DosQFileMode 180  
DosQFsAttach 182  
DosQFsInfo 185  
DosQHandType 187  
DosQNmPHandState 43  
DosQNmPipeInfo 45  
DosQPathInfo 189  
DosQSysInfo 193  
DosQVerify 194

Index

DosRawReadNmPipe 46  
DosRawWriteNmPipe 47  
DosRead 195  
DosReadAsync 197  
DosRmdir 200  
DosScanEnv 201  
DosSearchPath 202  
DosSelectDisk 205  
DosSetCp 71, 73  
DosSetFHandState 206  
DosSetFileInfo 209  
DosSetFileMode 211  
DosSetFsInfo 213  
DosSetMaxFH 215  
DosSetNmPHandState 48  
DosSetPathInfo 216  
DosSetVerify 219  
DosTransactNmPipe 50  
DosWaitNmPipe 51  
DosWrite 221  
DosWriteAsync 223  
drive, attach 233  
duplicate a file handle 127

+---+  
| E |  
+---+

EA 101  
EAOP  
fpGEAList  
fpFEAList offError  
end of process 249  
enumerate a directory's information 130  
enumerate a file's information 130  
enumerate a subdirectory's information 130  
environment segment, scan 201  
EOD 162  
extended attributes 101  
extended boot record 59  
Extended DOS Partition 59  
extended volume 59

Index

+---+  
| F |  
+---+

FAT type determination 56  
file (EOD) size, change 162, 283  
file access, lock 133, 137, 253  
file access, unlock 133, 137, 253  
file attributes 180, 212  
file handle 53  
file handle state, query 174  
file handle state, set 206  
file handle, close 118, 242  
file handle, commit 244  
file handle, duplicate 127  
file handles, set maximum 215  
file information, enumerate 130  
file information, query 177, 189  
    set 251, 291  
file information, set 209, 216  
file IO, read 133, 253  
file IO, write 133, 253  
file read  
    write position pointer 240  
file system control 155, 271  
file system ID, query 185  
file system information, query 185, 274  
file system information, set 213, 274  
file system name, query 185  
file, asynchronous read from a 197  
file, asynchronous write to 223  
file, change read  
    write pointer 116  
file, copy 119  
file, copy a 246  
file, create a 288  
file, delete a 248  
file, move a 160, 281  
file, move read  
    write pointer 116  
file, open 163  
file, open a 288  
file, query attribute 250  
file, query mode 180, 250  
file, read from a 195, 294  
file, rename a 160  
file, set attribute 211, 250  
file, set mode 211, 250  
file, write to a 221, 300

Index

filename 55  
filenames 293  
find close 256  
find first 257  
find first matching file 140  
find handle, close 139  
find next 261, 263  
find next matching file path name 146  
find-notify directory changes 149, 151, 266  
find-notify directory or file changes 268  
find-notify handle, close 148, 265  
flag process 25  
FSD attribute 105  
FSD initialization 106  
FSH\_ADDSHARE 304  
FSH\_BUFSTATE 306  
FSH\_CANONICALIZE 307  
FSH\_CHECKEANAME 308  
FSH\_CRITERORR 309  
FSH\_DEVIOCTL 311  
FSH\_DOVOLIO 313, 324  
FSH\_DOVOLIO2 316  
FSH\_FINDCHAR 318  
FSH\_FINDDUPHVPB 319  
FSH\_FLUSHBUF 320  
FSH\_FORCENOSWAP 321  
FSH\_GETBUF 322  
FSH\_GETFIRSTOVERLAPBUF 324  
FSH\_GETVOLPARM 326  
FSH\_INTERR 327  
FSH\_ISCURDIRPREFIX 328  
FSH\_LOADCHAR 329  
FSH\_NAMEFROMSFN 330  
FSH\_PREVCHAR 331  
FSH\_PROBEBUF 332  
FSH\_QSYSINFO 334  
FSH\_RELEASEBUF 335  
FSH\_REMOVESHARE 336  
FSH\_SEGALLOC 337  
FSH\_SEGFREE 339  
FSH\_SEGREALLOC 340  
FSH\_SEMCLEAR 341  
FSH\_SEMREQUEST 342  
FSH\_SEMSET 343  
FSH\_SEMSETWAIT 344  
FSH\_SEMWAIT 345  
FSH\_STORECHAR 346  
FSH\_UPPERCASE 347  
FSH\_WILDMATCH 348

EP 0 415 346 A2

Index

FSH\_YIELD 349  
FS\_ATTACH 233  
FS\_ATTRIBUTE 105  
FS\_ChDir 236  
FS\_CHGFILEPTR 240  
FS\_CLOSE 242  
FS\_COMMIT 244  
FS\_COPY 246  
FS\_DELETE 248  
FS\_EXIT 249  
FS\_FILEATTRIBUTE 250  
FS\_FILEINFO 251  
FS\_FILEIO 253  
FS\_FINDCLOSE 256  
FS\_FINDFIRST 257  
FS\_FINDFROMNAME 261  
FS\_FINDNEXT 263  
FS\_FINDNOTIFYCLOSE 265  
FS\_FindNotifyFirst 266  
FS\_FINDNOTIFYNEXT 268  
FS\_FLUSHBUF 270  
FS\_FSCTL 271  
FS\_FSINFO 274  
FS\_INIT 275  
FS\_IOCTL 276  
FS\_MKDIR 277  
FS\_MOUNT 278, 319  
FS\_MOVE 281  
FS\_NEWSIZE 283  
FS\_NMPIPE 284  
FS\_OPENCREATE 288  
FS\_PATHINFO 291  
FS\_PROCESSNAME 293  
FS\_READ 294  
FS\_RMDIR 296  
FS\_SETSWAP 297  
FS\_WRITE 300  
full EA  
    FEA 102  
    full extended attributes  
        FEA 102  
full EA list  
    FEAList 103

Index

+---+  
| G |  
+---+  
  
generic IOCTL 276  
Generic IOCTL, Category 8 66  
Get Device Parameters 66  
get EA  
    GEA 103  
    get extended attributes  
        GEA 102  
get EA list  
    GEAList 103  
Get Process Code Page 74  
  
+---+  
| H |  
+---+  
  
handle 53  
handle type, query 187  
  
+---+  
| I |  
+---+  
  
IFS CONFIG.SYS Function 108  
initialization 275  
initialization, system 82  
INT 21H  
    enumerate extended attributes 12  
    extended open  
        create 15  
    extended open2 15  
    get & set extended attributes 9  
    get & set media id 19  
    query DOS value 21  
    query max EA size supported 15  
IOCTL 58, 125, 276  
  
+---+  
| K |  
+---+  
  
Keyboard ANSI 53

Index

+---+  
| L |  
+---+

lock file access 133, 137, 253  
logical block device 59

+---+  
| M |  
+---+

make subdirectory 158, 277  
matching file name, find next 146  
message header, named pipe 36  
meta characters 129  
mount volume 278  
move a file 160, 281  
move a subdirectory 160, 281  
move file read  
    write pointer 116

+---+  
| N |  
+---+

name editing 129  
named pipe 284  
named pipe handle state, query 43  
named pipe handle state, set 48  
named pipe information, query 45  
named pipe message header 36  
named pipe, connect 32  
named pipe, create 34  
named pipe, direct read 46  
named pipe, direct write 47  
named pipe, disconnect 33  
named pipe, peek 40  
named pipe, procedure call 31  
named pipe, raw read 46  
named pipe, raw write 47  
named pipe, transact 50  
named pipe, wait 51

Index

377

Index

+---+  
| O |  
+---+

open a file 163, 288

+---+  
| P |  
+---+

path, search a 202  
pathname 55  
peek a named pipe 40  
pipe 28  
pipe, make 28  
procedure call via named pipe 31  
process, end of 249  
protect mode configuration 83  
pseudo device name 182  
pseudo-character device name 153, 182  
pseudo-character device name attached to an FSD, query 182  
pseudo-character device(s) 101

+---+  
| Q |  
+---+

query a directory's information 189  
query a file's information 177, 189  
query a handle type 187  
query a named pipe's information 45  
query a subdirectory's information 189  
query current directory 171  
query current disk 173  
query file attribute 250  
query file handle state 174  
query file mode 180, 250  
query file system information 185, 274  
query info about a pseudo-character device attached to an FSD 182  
query info about an attached remote file system 182  
query named pipe handle state 43  
query system variables 193  
query verify setting 194  
query  
    set a file's information 251  
    set a path or file's information 291

Index

00194 378

EP 0 415 346 A2



Index

+---+  
| R |  
+---+

raw mode 53  
raw read named pipe 46  
raw write named pipe 47  
read file IO 133, 253  
read from a file 195, 294  
real mode configuration 83  
recognition, volume 278  
remote named pipe operation 284  
remove subdirectory 200, 296  
rename a file 160  
rename a subdirectory 160  
reset, buffers 113, 244, 270

+---+  
| S |  
+---+

scan environment segment 201  
search a path 202  
search handle, close 139  
select disk drive 205  
set a directory's information 216  
set a file's information 209, 216  
set file attribute 211, 250  
set file handle state 206  
set file mode 211, 250  
set file system information 213, 274  
set maximum file handle 215  
set named pipe handle state 48  
set verify switch 219  
set volume label 213  
set volume serial number 213  
single file device 101  
speaker, sound 58  
subdirectory information, enumerate 130  
subdirectory information, query 189  
subdirectory, make 158, 277  
subdirectory, move a 160, 281  
subdirectory, remove 200, 296  
subdirectory, rename a 160  
swap file 297  
swapping 297  
system calls, FSD 106  
system configuration 83  
system initialization 82

Index

system variables, query 193

+---+  
| T |  
+---+

transact named pipe 50

+---+  
| U |  
+---+

Uniform Naming Convention (UNC) 55  
unlock file access 133, 137, 253

+---+  
| V |  
+---+

Verify directory path 236  
verify setting, query 194  
verify switch, set 219  
Video ANSI 53  
volume label, query 185  
volume label, set 213  
volume recognition 278  
volume serial number, query 185  
volume serial number, set 213  
volume, mount 278

+---+  
| W |  
+---+

wait named pipe 51  
write file IO 133, 253  
write to a file 221, 300

+---+  
| Y |  
+---+

yield 349

## APPENDIX II

```

/*static char *SCCSID = "@(#)fsd.h 1.31 89/08/03 */
5 /* fsd.h - File system driver entry point declarations */

/* FS_ATTRIBUTE bit field values */

#define FSA_REMOTE 0x00000001 /* Set if REMOTE FSD */
10 #define FSA_UNC 0x00000002 /* Set if FSD implements UNC support */
#define FSA_LOCK 0x00000004 /* Set if FSD needs lock notification */

#define CDDWORKAREASIZE 8
#define SFDWORKAREASIZE 30
15 #define VPDWORKAREASIZE 36

/* Volume Parameter structures */

20 #define VPBTEXTLEN 12

struct vpfsi {
 unsigned long vpi_vid; /* 32 bit volume ID */
 unsigned long vpi_hDEV; /* handle to device driver */
25 unsigned short vpi_bsize; /* sector size in bytes */
 unsigned long vpi_totsec; /* total number of sectors */
 unsigned short vpi_trksec; /* sectors / track */
 unsigned short vpi_nhead; /* number of heads */
 char vpi_text[VPBTEXTLEN]; /* volume name */
30 }; /* vpfsi */

/*
 * Predefined volume IDs: [note - keep this list in sync with list in
 * dos/dosinc/vpb.inc!]
35 */
/* Unique ID for vpb_ID field for unreadable media. */
#define UNREAD_ID 0x534E4A52L /* Stored as (bytes) 0x52,4A,4E,53. */

/*
40 * Unique ID for vpb_ID field for damaged volume (recognized by IFS but cannot
 * be normally mounted).
 */
#define DAMAGED_ID 0x0L /* Stored as (bytes) 0,0,0,0. */

45 /* file system dependent - volume params */
struct vpfsd {
 char vpd_work[VPDWORKAREASIZE]; /* work area */
}; /* vpfsd */

50 /* Current Directory structures */

struct cdfsi {
 unsigned short cdi_hVPB; /* VPB handle for associated device */
55 unsigned short cdi_end; /* end of root portion of curdir */
 char cdi_flags; /* flags indicating state of cdfsd */

```

```

 char cdi_curdir[CCHMAXPATH]; /* text of current directory */
}; /* cdfsi */

/* bit values for cdi_flags (state of cdfsd structure */
5
#define CDI_ISVALID 0x80 /* format is known */
#define CDI_ISROOT 0x40 /* cur dir == root */
#define CDI_ISCURRENT 0x20

10 struct cdfsd {
 char cdd_work[CDDWORKAREASIZE]; /* work area */
}; /* cdfsd */

15 /* Per open-instance (System File) structures */

struct sffsi {
 unsigned long sfi_mode; /* access/sharing mode */
 unsigned short sfi_hVPB; /* volume info. */
20 unsigned short sfi_ctime; /* file creation time */
 unsigned short sfi_cdate; /* file creation date */
 unsigned short sfi_atime; /* file access time */
 unsigned short sfi_adata; /* file access date */
 unsigned short sfi_mtime; /* file modification time */
25 unsigned short sfi_mdate; /* file modification date */
 unsigned long sfi_size; /* size of file */
 unsigned long sfi_position; /* read/write pointer */
 /* the following may be of use in sharing checks */
 unsigned short sfi_UID; /* user ID of initial opener */
30 unsigned short sfi_PID; /* process ID of initial opener */
 unsigned short sfi_PDB; /* PDB (in 3.x box) of initial opener */
 unsigned short sfi_selfsfn; /* system file number of file instance */
 unsigned char sfi_tstamp; /* update/propagate time stamp flags */
 /* use with ST_Sxxx and ST_Pxxx */
35 unsigned short sfi_type; /* use with STYPE_ */
}; /* sffsi */

/* sfi_tstamps flags */
#define ST_SCREAT 1 /* stamp creation time */
40 #define ST_PCREAT 2 /* propagate creation time */
#define ST_SWRITE 4 /* stamp last write time */
#define ST_PWRITE 8 /* propagate last write time */
#define ST_SREAD 16 /* stamp last read time */
#define ST_PREAD 32 /* propagate last read time */
45

/* sfi_type flags */
#define STYPE_FILE 0 /* file */
#define STYPE_DEVICE 1 /* device */
#define STYPE_NMPIPE 2 /* named pipe */
50 #define STYPE_FCB 4 /* fcb sft */

/* file system dependent - file instance */
struct sffsd {
 char sfd_work[SFDWORKAREASIZE]; /* work area */
55 }; /* sffsd */

```

```

/* file system independent - file search parameters */
struct fsfsi {
5 unsigned short fsi_hVPB; /* volume info. */
}; /* fsfsi */

/* file system dependent - file search parameters */
#define FSFSD_WORK_SIZE 24
struct fsfsd {
10 char fsd_work[FSFSD_WORK_SIZE]; /* work area */
}; /* fsfsd */

/* file system dependent - device information */
struct devfsd {
15 unsigned long FSDRef; /* Reference obtained from FSD during ATTACH
*/
}; /* devfsd */

/*****
20 *
* union and structures for FS_FSCTL
*/
/* pArgType == 1, FileHandle directed case */
struct SF {
25 struct sffsi far * psffsi;
 struct sffsd far * psffsd;
}; /* SF */

/* pArgType == 2, PathName directed case */
30 struct CD {
 struct cdfsi far * pcdfsi;
 struct cdfsd far * pcdfsd;
 char far * pPath;
 unsigned short iCurDirEnd;
35 }; /* CD */

union argdat {
/* pArgType == 1, FileHandle directed case */
struct SF sf;
40

/* pArgType == 2, PathName directed case */
struct CD cd;

/* pArgType == 3, FSD Name directed case */
45 /* nothing */
}; /* argdat */

/*****
50 *
* Union and structures for FS_NMPIPE
*
*/

55 /* Get/SetPHandState parameter block */
struct phs_param {

```

```

 short phs_len;
 short phs_dlen;
 short phs_pmode; /* pipe mode set or returned */
};
5

/* DosQNmPipeInfo parameter block,
 * data is info. buffer addr */
struct npi_param {
10 short npi_len;
 short npi_dlen;
 short npi_level; /* information level desired */
};

15

/* DosRawReadNmPipe parameters,
 * data is buffer addr */
struct npr_param {
20 short npr_len;
 short npr_dlen;
 short npr_nbyt; /* number of bytes read */
};

/* DosRawWriteNmPipe parameters,
 * data is buffer addr */
25 struct npw_param {
 short npw_len;
 short npw_dlen;
 short npw_nbyt; /* number of bytes written */
30 };

/* NPipeWait parameters */
struct npq_param {
35 short npq_len;
 short npq_dlen;
 long npq_timeo; /* timeout in milliseconds */
 short npq_prio; /* priority of caller */
};

40 /* DosCallNmPipe parameters,
 * data is in-buffer addr */
struct npx_param {
 short npx_len;
 unsigned short npx_ilen; /* length of in-buffer */
45 char far *npx_obuf; /* pointer to out-buffer */
 unsigned short npx_olen; /* length of out-buffer */
 unsigned short npx_nbyt; /* number of bytes read */
 long npx_timeo; /* timeout in milliseconds */
50 };

/* PeekPipe parameters, data is buffer addr */
struct npp_param {
 short npp_len;
 unsigned short npp_dlen;
55 unsigned short npp_nbyt; /* number of bytes read */
 unsigned short npp_avl0; /* bytes left in pipe */
};

```

```

 unsigned short npp_avl1; /* bytes left in current msg */
 unsigned short npp_state; /* pipe state */
 };

5 /* DosTransactNmPipe parameters,
 * data is in-buffer addr */
 struct npt_param {
 short npt_len;
 unsigned short npt_ilen; /* length of in-buffer */
10 char far *npt_obuf; /* pointer to out-buffer */
 unsigned short npt_oien; /* length of out-buffer */
 unsigned short npt_nbyt; /* number of bytes read */
 };

15 /* QNmpipeSemState parameter block,
 * data is user data buffer */
 struct qnps_param {
 unsigned short qnps_len; /* length of parameter block */
 unsigned short qnps_dlen; /* length of supplied data block */
20 long qnps_semh; /* system semaphore handle */
 unsigned short qnps_nbyt; /* number of bytes returned */
 };

 /* ConnectPipe parameter block, no data block */
25 struct npc_param {
 unsigned short npc_len; /* length of parameter block */
 unsigned short npc_dlen; /* length of data block */
 };

30 /* DisconnectPipe parameter block, no data block */
 struct npd_param {
 unsigned short npd_len; /* length of parameter block */
 unsigned short npd_dlen; /* length of data block */
 };

35 union npoper {
 struct phs_param phs;
 struct npi_param npi;
 struct npr_param npr;
40 struct npw_param npw;
 struct npq_param npq;
 struct npx_param npx;
 struct npp_param npp;
 struct npt_param npt;
45 struct qnps_param qnps;
 struct npc_param npc;
 struct npd_param npd;
 }; /* npoper */

50 /*****
 *
 * Declarations for the FSD entry points.
 *
55 */

```

```

/* bit values for the IOflag parameter in various FS_ entry points */
#define IOFL_WRITETHRU 0x10 /* Write through bit */
#define IOFL_NOCACHE 0x20 /* No Cache bit */

5
int far pascal
FS_ATTACH(
 unsigned short, /* flag */
 char far *, /* pDev */
10 void far *, /* if remote drive
 struct vpfds far *
 else if remote device
 null ptr (0L) */
 void far *, /* if remote drive
 struct cdfs far *
 else
 struct devfsd far * */
15 char far *, /* pParm */
 unsigned short far * /* pLen */
20);

/* values for flag in FS_ATTACH */
#define FSA_ATTACH 0x00
#define FSA_DETACH 0x01
25 #define FSA_ATTACH_INFO 0x02

int far pascal
FS_CHDIR(
 unsigned short, /* flag */
30 struct cdfs far *, /* pcdfsi */
 struct cdfs far *, /* pcdfsd */
 char far *, /* pDir */
 unsigned short /* iCurDirEnd */
35);

/* values for flag in FS_CHDIR */
#define CD_EXPLICIT 0x00
#define CD_VERIFY 0x01
40 #define CD_FREE 0x02

int far pascal
FS_CHGFILEPTR(
 struct sffsi far *, /* psffsi */
 struct sffsd far *, /* psffsd */
45 long, /* offset */
 unsigned short, /* type */
 unsigned short /* IOflag */
50);

/* values for type in FS_CHGFILEPTR */
#define CFP_RELBEGIN 0x00
#define CFP_RELCUR 0x01
#define CFP_RELEND 0x02

55 int far pascal
FS_CLOSE(

```

```

 unsigned short, /* close type */
 unsigned short, /* IOflag */
 struct sfsi far *, /* psfsi */
 struct sfsd far * /* psfsd */
5);
 #define FS_CL_ORDINARY 0
 /* ordinary close of file */
 #define FS_CL_FORPROC 1
 /* final close of file for the process */
10 #define FS_CL_FORSYS 2
 /* final close of file for the system (for all processes) */

int far pascal
FS_COMMIT(
15 unsigned short, /* commit type */
 unsigned short, /* IOflag */
 struct sfsi far *, /* psfsi */
 struct sfsd far * /* psfsd */
);
20 /* values for commit type */
 #define FS_COMMIT_ONE 1
 /* commit for a single file */
 #define FS_COMMIT_ALL 2
 /* commit due to buf reset - for all files */
25

int far pascal
FS_COPY(
 unsigned short, /* copy mode */
 struct cfsi far *, /* pcfsi */
30 struct cfsd far *, /* pcfsd */
 char far *, /* source name */
 unsigned short, /* iSrcCurrDirEnd */
 char far *, /* pDst */
 unsigned short, /* iDstCurrDirEnd */
35 unsigned short /* nameType (flags) */
);

int far pascal
FS_DELETE(
40 struct cfsi far *, /* pcfsi */
 struct cfsd far *, /* pcfsd */
 char far *, /* pFile */
 unsigned short /* iCurDirEnd */
);
45

void far pascal
FS_EXIT(
 unsigned short, /* uid */
 unsigned short, /* pid */
50 unsigned short /* pdb */
);

int far pascal
FS_FILEATTRIBUTE(
55 unsigned short, /* flag */
 struct cfsi far *, /* pcfsi */

```



```

 struct cdfsd far *, /* pcdfsd */
 char far *, /* pName */
 unsigned short, /* iCurDirEnd */
 unsigned short far * /* pAttr */
5);

/* values for flag in FS_FILEATTRIBUTE */
#define FA_RETRIEVE 0x00
#define FA_SET 0x01
10

int far pascal
FS_FILEINFO(
 unsigned short, /* flag */
 struct sffsi far *, /* psffsi */
15 struct sffsd far *, /* psffsd */
 unsigned short, /* level */
 char far *, /* pData */
 unsigned short, /* cbData */
 unsigned short /* IOflag */
20);

/* values for flag in FS_FILEINFO */
#define FI_RETRIEVE 0x00
#define FI_SET 0x01
25

int far pascal
FS_FILEIO(
 struct sffsi far *, /* psffsi */
 struct sffsd far *, /* psffsd */
30 char far *, /* cbCmdList */
 unsigned short, /* pCmdLen */
 unsigned short far *, /* poError */
 unsigned short /* IOflag */
35);

int far pascal
FS_FINDCLOSE(
 struct fsfsi far *, /* pfsfsi */
 struct fsfsd far * /* pfsfsd */
40);

/* values for flag in FS_FindFirst, FS_FindFromName, FS_FindNext */
#define FF_NOPOS 0x00
#define FF_GETPOS 0x01
45

int far pascal
FS_FINDFIRST(
 struct cdfsi far *, /* pcdfsi */
50 struct cdfsd far *, /* pcdfsd */
 char far *, /* pName */
 unsigned short, /* iCurDirEnd */
 unsigned short, /* attr */
 struct fsfsi far *, /* pfsfsi */
55 struct fsfsd far *, /* pfsfsd */
 char far *, /* pData */

```

```

 unsigned short, /* cbData */
 unsigned short far *, /* pcMatch */
 unsigned short, /* level */
 unsigned short /* flags */
);

```

```

int far pascal
FS_FINDFROMNAME(
 struct fsfsi far *, /* pfsfsi */
 struct fsfsd far *, /* pfsfsd */
 char far *, /* pData */
 unsigned short, /* cbData */
 unsigned short far *, /* pcMatch */
 unsigned short, /* level */
 unsigned long, /* position */
 char far *, /* pName */
 unsigned short /* flags */
);

```

```

int far pascal
FS_FINDNEXT(
 struct fsfsi far *, /* pfsfsi */
 struct fsfsd far *, /* pfsfsd */
 char far *, /* pData */
 unsigned short, /* cbData */
 unsigned short far *, /* pcMatch */
 unsigned short, /* level */
 unsigned short /* flag */
);

```

```

int far pascal
FS_FINDNOTIFYCLOSE(
 unsigned short /* handle */
);

```

```

int far pascal
FS_FINDNOTIFYFIRST(
 struct cdfsi far *, /* pcdfsi */
 struct cdfsd far *, /* pcdfsd */
 char far *, /* pName */
 unsigned short, /* iCurDirEnd */
 unsigned short, /* attr */
 unsigned short far *, /* pHandle */
 char far *, /* pData */
 unsigned short, /* cbData */
 unsigned short far *, /* pcMatch */
 unsigned short, /* level */
 unsigned long /* timeout */
);

```

```

int far pascal
FS_FINDNOTIFYNEXT(
 unsigned short, /* handle */
 char far *, /* pData */
 unsigned short, /* cbData */
 unsigned short far *, /* pcMatch */

```

```

 unsigned short, /* infolevel */
 unsigned long /* timeout */
);

int far pascal
FS_FLUSHBUF(
 unsigned short, /* hVPB */
 unsigned short /* flag */
);

/* values for flag in FS_FLUSH */
#define FLUSH_RETAIN 0x00
#define FLUSH_DISCARD 0x01

int far pascal
FS_FSCTL(
 union argdat far *, /* pArgdat */
 unsigned short, /* iArgType */
 unsigned short, /* func */
 char far *, /* pParm */
 unsigned short, /* lenParm */
 unsigned short far *, /* plenParmOut */
 char far *, /* pData */
 unsigned short, /* lenData */
 unsigned short far * /* plenDataOut */
);

/* values for iArgType in FS_FSCTL */
#define FSCTL_ARG_FILEINSTANCE 0x01
#define FSCTL_ARG_CURDIR 0x02
#define FSCTL_ARG_NULL 0x03

/* values for func in FS_FSCTL */
#define FSCTL_FUNC_NONE 0x00
#define FSCTL_FUNC_NEW_INFO 0x01
#define FSCTL_FUNC_EASIZE 0x02

int far pascal
FS_FSINFO(
 unsigned short, /* flag */
 unsigned short, /* hVPB */
 char far *, /* pData */
 unsigned short, /* cbData */
 unsigned short /* level */
);

/* values for flag in FS_FSINFO */
#define INFO_RETRIEVE 0x00
#define INFO_SET 0x01

int far pascal
FS_INIT(
 char far *, /* szParm */
 unsigned long, /* pDevHlp */
 unsigned long far * /* pMiniFSD */
);

```

```

int far pascal
FS_IOCTL(
 struct sffsi far *, /* psffsi */
 struct sffsd far *, /* psffsd */
 unsigned short, /* cat */
 unsigned short, /* func */
 char far *, /* pParm */
 unsigned short, /* lenParm */
 char far *, /* pData */
 unsigned short /* lenData */
);

int far pascal
FS_MKDIR(
 struct cdlsi far *, /* pcdfsi */
 struct cdfsd far *, /* pcdfsd */
 char far *, /* pName */
 unsigned short, /* iCurDirEnd */
 char far *, /* pEABuf */
 unsigned short /* flags */
);

int far pascal
FS_MOUNT(
 unsigned short, /* flag */
 struct vpfsi far *, /* pvpfsi */
 struct vpfds far *, /* pvpfsd */
 unsigned short, /* hVPB */
 char far *, /* pBoot */
 /* values for flag in FS_MOUNT */
 #define MOUNT_MOUNT 0x00
 #define MOUNT_VOL_REMOVED 0x01
 #define MOUNT_RELEASE 0x02
 #define MOUNT_ACCEPT 0x03
);

int far pascal
FS_MOVE(
 struct cdlsi far *, /* pcdfsi */
 struct cdfsd far *, /* pcdfsd */
 char far *, /* pSrc */
 unsigned short, /* iSrcCurDirEnd */
 char far *, /* pDst */
 unsigned short, /* iDstCurDirEnd */
 unsigned short /* flags */
);

int far pascal
FS_NEWSIZE(
 struct sffsi far *, /* psffsi */
 struct sffsd far *, /* psffsd */
 unsigned long, /* len */
 unsigned short /* IOflag */
);

```

```

int far pascal
FS_NMPIPE(
5 struct sffsi far *, /* psffsi */
 struct sffsd far *, /* psffsd */
 unsigned short, /* OpType */
 union npoper far *, /* pOpRec */
 char far *, /* pData */
10 char far *, /* pName */
);

/* Values for OpType in FS_NMPIPE */

15 #define NMP_GetPHandState 0x21
 #define NMP_SetPHandState 0x01
 #define NMP_PipeQInfo 0x22
 #define NMP_PeekPipe 0x23
 #define NMP_ConnectPipe 0x24
20 #define NMP_DisconnectPipe 0x25
 #define NMP_TransactPipe 0x26
 #define NMP_READRAW 0x11
 #define NMP_WRITERAW 0x31
 #define NMP_WAITPIPE 0x53
25 #define NMP_CALLPIPE 0x54
 #define NMP_QNmPipeSemState 0x58

int far pascal
30 FS_OPENCREATE(
 struct cdfsi far *, /* pcdfsi */
 void far *, /* if remote device
 struct devfsd far *
 else
35 struct cdfsd far */
 char far *, /* pName */
 unsigned short, /* iCurDirEnd */
 struct sffsi far *, /* psffsi */
 struct sffsd far *, /* psffsd */
40 unsigned short, /* fhandflag */
 unsigned short, /* openflag */
 unsigned short far *, /* pAction */
 unsigned short, /* attr */
 char far *, /* pEABuf */
45 unsigned short far *, /* pfgenFlag */
);

#define FOC_NEEDEAS 0x1 /*there are need eas for this file */

50 int far pascal
 FS_PATHINFO(
 unsigned short, /* flag */
 struct cdfsi far *, /* pcdfsi */
 struct cdfsd far *, /* pcdfsd */
55 char far *, /* pName */
 unsigned short, /* iCurDirEnd */

```

```

 unsigned short, /* level */
 char far *, /* pData */
 unsigned short /* cbData */
);
5
 /* values for flag in FS_PATHINFO */
 #define PI_RETRIEVE 0x00
 #define PI_SET 0x01

10 int far pascal
 FS_PROCESSNAME(
 char far * /* pNameBuf */
);

15 int far pascal
 FS_READ(
 struct sffsi far *, /* psffsi */
 struct sffsd far *, /* psffsd */
 char far *, /* pData */
20 unsigned short far *, /* pLen */
 unsigned short /* IOflag */
);

 int far pascal
25 FS_RMDIR(
 struct cdfsi far *, /* pcdfsi */
 struct cdfsd far *, /* pcdfsd */
 char far *, /* pName */
 unsigned short /* iCurDirEnd */
30);

 int far pascal
 FS_SETSWAP(
 struct sffsi far *, /* psffsi */
35 struct sffsd far * /* psffsd */
);

 int far pascal
 FS_SHUTDOWN(
40 unsigned short, /* usType */
 unsigned long /* ulReserved */
);

 /* values for usType in FS_SHUTDOWN */
45 #define SD_BEGIN 0x00
 #define SD_COMPLETE 0x01

 int far pascal
 FS_WRITE(
50 struct sffsi far *, /* psffsi */
 struct sffsd far *, /* psffsd */
 char far *, /* pData */
 unsigned short far *, /* pLen */
 unsigned short /* IOflag */
55);

```

```

int far pascal
MFS_CHGFILEPTR(
 unsigned long, /* offset */
 unsigned short /* type */
5);

int far pascal
MFS_CLOSE(
10 void
);

int far pascal
MFS_INIT(
15 void far *, /* bootdata */
 char far *, /* number io */
 long far *, /* vectorripl */
 void far *, /* bpb */
 unsigned long far *, /* pMiniFSD */
20 unsigned long far * /* dump address */
);

int far pascal
MFS_OPEN(
25 char far *, /* name */
 unsigned long far * /* size */
);

int far pascal
30 MFS_READ(
 char far *, /* data */
 unsigned short far * /* length */
);

35 int far pascal
MFS_TERM(
 void
);

```

## APPENDIX III

```

/*static char *SCCSID = "@(#)fsh.h 1.16 89/04/18";*/
5 /* fsh.h - FSH_ = fshelper interface declarations */

/*
 * FSH_DOVOLIO2 omits flag definition
 *
10 * Since we are using C5.1, the function prototypes should be made to
 * conform with true ANSI standards. I have converted FSH_ADDSHARE
 * as an example.
 *
 */
15 #if 1
 int far pascal
 FSH_ADDSHARE(
 char far *, /* pName */
20 unsigned short, /* mode */
 unsigned short, /* hVPB */
 unsigned long far * /* phShare */
);
 #else
25 USHORT far pascal
 FSH_ADDSHARE(
 PSZ pName,
 USHORT mode,
 SHANDLE hVPB,
30 LHANDLE phShare
);
 #endif

 int far pascal
35 FSH_BUFSTATE(
 char far *, /* pBuf */
 unsigned short, /* flag */
 unsigned short far * /* pState */
);
40 int far pascal
 FSH_CANONICALIZE(
 char far *, /* pPathName */
 unsigned short, /* cbPathBuf */
45 char far * /* pPathBuf */
);

 int far pascal
 FSH_CHECKEANAME(
50 unsigned short, /* level */
 unsigned long, /* len of name */
 char far * /* pEAName */
);

55 int far pascal
 FSH_CRITERROR(

```



```

 int, /* cbMessage */
 char far *, /* pMessage */
 int, /* nSubs */
 char far *, /* pSubs */
5 unsigned short /* fAllowed */
);

/* Flags for fAllowed
 */
10 #define CE_ALLFAIL 0x0001 /* FAIL allowed */
 #define CE_ALLABORT 0x0002 /* ABORT allowed */
 #define CE_ALLRETRY 0x0004 /* RETRY allowed */
 #define CE_ALLIGNORE 0x0008 /* IGNORE allowed */
 #define CE_ALLACK 0x0010 /* ACK allowed */
15 /* Return values from FSH_CRITERR
 */
 #define CE_RETIGNORE 0x0000 /* User said IGNORE */
 #define CE_RETRETRY 0x0001 /* User said RETRY */
20 #define CE_RETFAIL 0x0003 /* User said FAIL/ABORT */
 #define CE_RETACK 0x0004 /* User said continue */

int far pascal
FSH_DEVIOCTL(
25 unsigned short, /* FSDRaisedFlag */
 unsigned long, /* hDev */
 unsigned short, /* sfn */
 unsigned short, /* cat */
 unsigned short, /* func */
30 char far *, /* pParm */
 unsigned short, /* cbParm */
 char far *, /* pData */
 unsigned short /* cbData */
);
35 int far pascal
FSH_DOVOLIO(
 unsigned short, /* operation */
 unsigned short, /* fAllowed */
40 unsigned short, /* hVPB */
 char far *, /* pData */
 unsigned short far *, /* pcSec */
 unsigned long /* iSec */
);
45 /* Flags for operation
 */
 #define DVIO_OPREAD 0x0000 /* no bit on => readi */
 #define DVIO_OPWRITE 0x0001 /* ON => write else read */
50 #define DVIO_OPBYPASS 0x0002 /* ON => cache bypass else no bypass */
 #define DVIO_OPVERIFY 0x0004 /* ON => verify after write */
 #define DVIO_OPHARDERR 0x0008 /* ON => return hard errors directly */
 #define DVIO_OPWRTHRU 0x0010 /* ON => write thru */
55 #define DVIO_OPNCACHE 0x0020 /* ON => don't cache data */

```

```

/* Flags for fAllowed
*/
#define DVIO_ALLFAIL 0x0001 /* FAIL allowed */
#define DVIO_ALLABORT 0x0002 /* ABORT allowed */
5 #define DVIO_ALLRETRY 0x0004 /* RETRY allowed */
#define DVIO_ALLIGNORE 0x0008 /* IGNORE allowed */
#define DVIO_ALLACK 0x0010 /* ACK allowed */

int far pascal
10 FSH_DOVOLIO2(
 unsigned long, /* hDev */
 unsigned short, /* sfn */
 unsigned short, /* cat */
 unsigned short, /* func */
15 char far *, /* pParm */
 unsigned short, /* cbParm */
 char far *, /* pData */
 unsigned short /* cbData */
);
20
int far pascal
FSH_FINDCHAR(
 unsigned short, /* nChars */
 char far *, /* pChars */
25 char far * far * /* ppStr */
);

int far pascal
FSH_FINDDUPHVPB(
30 unsigned short, /* hVPB */
 unsigned short far * /* pHVPB */
);

int far pascal
35 FSH_FLUSHBUF(
 unsigned short, /* hVPB */
 unsigned short /* fDiscard */
);
40 /* fDiscard values
*/
#define FB_DISCNONE 0x0000 /* Do not discard buffers */
#define FB_DISCCLEAN 0x0001 /* Discard clean buffers */

45 int far pascal
FSH_FORCENOSWAP(
 unsigned short /* sel */
);
50 int far pascal
FSH_GETBUF(
 unsigned long, /* iSec */
 unsigned short, /* fPreRead */
 unsigned short, /* hVPB */
55 char far * far * /* ppBuf */
);

```

```

/* Flags for fPreRead
*/
#define GB_PRNOREAD 0x0001 /* ON => no preread occurs */

5
int far pascal
FSH_GETOVERLAPBUF(
 unsigned short, /* hVPB */
 unsigned long, /* iSec */
10 unsigned long, /* iSec */
 unsigned long far *, /* piSecBuf */
 char far * far * /* ppBuf */
);

15 int far pascal
FSH_GETVOLPARM(
 unsigned short, /* hVPB */
 struct vpfsi far * far *, /* ppVPBfsi */
 struct vpfsd far * far *, /* ppVPBfsd */
20);

int far pascal
FSH_INTERR(
 char far *, /* pMsg */
25 unsigned short /* cbMsg */
);

int far pascal
FSH_ISCURDIRPREFIX(
30 char far * /* pName */
);

void far pascal
FSH_LOADCHAR(
35 char far * far *, /* ppStr */
 unsigned short far * /* pChar */
);

void far pascal
40 FSH_PREVCHAR(
 char far *, /* pBeg */
 char far * far * /* ppStr */
);

45 int far pascal
FSH_PROBEBUF(
 unsigned short, /* operation */
 char far *, /* pData */
 unsigned short /* cbData */
50);

/* Values for operation
*/
#define PB_OPREAD 0x0000 /* Check for read access */
55 #define PB_OPWRITE 0x0001 /* Check for write access */

```

```

int far pascal
FSH_QSYSINFO(
 unsigned short, /* index */
 char far *, /* pData */
 unsigned short /* cbData */
);

/* Values for index
 */
#define QSI_SECSIZE 1 /* index to query max sector size */
#define QSI_PROCID 2 /* index to query PID,UserID and Currentpdb */
#define QSI_THREADNO 3 /* index to query abs.thread no */
#define QSI_VERIFY 4 /* index to query per-process verify */

int far pascal
FSH_NAMEFROMSFN(
 unsigned short, /* sfn */
 char far *, /* pName */
 unsigned short far * /* pcbName */
);

int far pascal
FSH_RELEASEBUF(void);

int far pascal
FSH_REMOVESHARE(
 unsigned long /* hShare */
);

int far pascal
FSH_SEGALLOC(
 unsigned short, /* flags */
 unsigned long, /* cbSeg */
 unsigned short far * /* pSel */
);

/* Fields for flags
 */
#define SA_FLDT 0x0001 /* ON => alloc LDT else GDT */
#define SA_FSWAP 0x0002 /* ON => swappable memory */
#define SA_FRINGMASK 0x6000 /* mask for isolating ring */
#define SA_FRING0 0x0000 /* ring 0 */
#define SA_FRING1 0x2000 /* ring 1 */
#define SA_FRING2 0x4000 /* ring 2 */
#define SA_FRING3 0x6000 /* ring 3 */

int far pascal
FSH_SEGFREE(
 unsigned short /* sel */
);

int far pascal
FSH_SEGREALLOC(
 unsigned short, /* sel */

```

```

 unsigned long /* cbSeg */
);

5 /* Timeout equates for all semaphore operations
 */
 #define TO_INFINITE 0xFFFFFFFFL
 #define TO_NOWAIT 0x00000000L

10 int far pascal
 FSH_SEMCLEAR(
 void far * /* pSem */
);

15 int far pascal
 FSH_SEMREQUEST(
 void far *, /* pSem */
 unsigned long /* cmsTimeout */
);

20 int far pascal
 FSH_SEMSET(
 void far * /* pSem */
);

25 int far pascal
 FSH_SEMSETWAIT(
 void far *, /* pSem */
 unsigned long /* cmsTimeout */
);

30);

 int far pascal
 FSH_SEMWAIT(
 void far *, /* pSem */
 unsigned long /* cmsTimeout */
);

35);

 int far pascal
 FSH_STORECHAR(
40 unsigned short, /* chDBCS */
 char far * far * /* ppStr */
);

 int far pascal
45 FSH_UPPERCASE(
 char far *, /* pName */
 unsigned short, /* cbPathBuf */
 char far * /* pPathBuf */
);

50);

 int far pascal
 FSH_WILDMATCH(
 char far *, /* pPat */
 char far * /* pStr */
55);

```

```

int far pascal
FSH_YIELD(void);

int far pascal
5 MFSH_DOVOLIO(
 char far *, /* Data */ /* */
 unsigned short far *, /* cSec */ /* */
 unsigned long /* iSec */ /* */
);
10
int far pascal
MFSH_INTERR(
 char far *, /* Msg */ /* */
 unsigned short /* cbMsg */ /* */
15);

int far pascal
MFSH_SEGALLOC(
 unsigned short, /* Flag */ /* */
 unsigned long, /* cbSeg */ /* */
 unsigned short far * /* Sel */ /* */
20);

int far pascal
25 MFSH_SEGFREE(
 unsigned short /* Sel */ /* */
);

int far pascal
30 MFSH_SEGREALLOC(
 unsigned short, /* Sel */ /* */
 unsigned long /* cbSeg */ /* */
);

35 int far pascal
MFSH_CALLRM(
 unsigned long far * /* Proc */ /* */
);

40 int far pascal
MFSH_LOCK(
 unsigned short, /* Sel */ /* */
 unsigned long far * /* Handle */ /* */
);
45
int far pascal
MFSH_PHYSTOVIRT(
 unsigned long, /* Addr */ /* */
 unsigned short, /* Len */ /* */
 unsigned short far * /* Sel */ /* */
50);

int far pascal
MFSH_UNLOCK(
55 unsigned long /* Handle */ /* */
);

```

```

int far pascal
MFSH_UNPHYSTOVIRT(
5 unsigned short /* Sel */
);

int far pascal
MFSH_VIRT2PHYS(
10 unsigned long, /* VirtAddr */
 unsigned long far * /* PhysAddr */
);

```

## APPENDIX IV

```

/**** BASEFSD.C
*
* IFS component test FSD
5 * Copyright 1988, Microsoft Corp.
* Sue Adams
*
* Description:
* Each entry point in this FSD checks its own tid table to see whether
10 * or not to save its parameters on behalf of the currently executing
* thread.
*
* Use:
* The makefile should define a different symbol for each FSD created
15 * from this source file, so that different FSNAME'S may be used.
*
* Special Information:
* =====
* A table is kept for each thread to place a data buffer (GDT) selector, if
20 * needed.
* Since the thread number is an absolute system thread number, no semaphore
* is needed on the selector, since a thread can only be executing this code
* in one place at a time. Each thread is limited to one GDT, and when
* finished with it, is expected to free it with the FS_FSCTL entry.
25 *
* This implementation keeps one table per/FS_ entry for enabling/disabling
* the saving of incoming parameters on behalf of a thread.
*
* See TABLE.C for table management routines.
30 *
* Warning
* =====
* The FSD will be in an inconsistent state if the system is shutdown via
* DOSSHUTDOWN. The system must be rebooted if other tests are to be run
35 * using this FSD after the shutdown test. Reason: The flag which allows
* the FSD to distinguish the shutdown test from other tests cannot be reset
* after shutdown. There is no way to communicate with the FSD via FS_FSCTL
* once a DOSSHUTDOWN call is made. This is per 1.2 dcr 259, and as of yet
* has not been decided if FS_FSCTL access will be allowed or not. If it
40 * becomes the case that FSCTL is accessible, then the 'pSDdata' as well as
* 'Is_Shutdown_Test' flag can be reset to 0 after the shutdown test. If it
* is decided that shutdown will be reversible, then SD_Status must be reset
* as well.
*
45 * Enforced conventions:
* =====
* The application must enable the appropriate FS_ entry point using
* DOSFSCTL before each FS api call. The entry points will take
* care of disabling themselves each time they are called.
50 *
* After an application makes an API call
* that will enter this fsd, if the entry point puts a GDT selector into the
* selector table on behalf of the thread, a subsequent call to DOSFSCTL
* must be made to free the selector and table entry.
55 *

```



```

* Modification History:
* 89.07.10 -- took out ifdefs for DCR509 -- changes enabled in kernel
* version 12.102 (new flags parameter to some entry pts.)
* 89.06.23 -- added ifdefs for DCR509
* 89.06.20 -- added global OpenAction for FS_OPENCREATE to use
* 89.05.17 -- added support for 1.2 DCR 508: pflag param to
* FS_OPENCREATE
* 89.04.07 -- integrated basefsd.c and base2fsd.c into one source
* 89.03.27 -- added define for CCHMAXPATH
* 89.02.02 -- added parameter-save-enable capabilities
* 89.01.09 -- initial version
*/

#include <doscalls.h>
#define CCHMAXPATH MAXPATHLEN /* so dont have to include bsedos.h */
#include <fsd.h>
#include <fsh.h>
#include <error.h>
#include "boot.h"
#include "basefsd.h"
#include "table.h"

/*
* Global Definitions
* =====
*/
#define NULLFEACBLIST (sizeof(((struct FEAList *)0)->feal_cbList))
#define MAXFEACBLIST 2520 /* 2520 x 26 (drive letters) fits in 1 seg */
#define CalcFEAoff(d) ((unsigned)((char)d - 'A') * (unsigned)MAXFEACBLIST)
#define NUMDRIVES 26
/*
* Forward Declarations
* =====
*/
int Init_FEA_Tab();
int DelFEAList(char);
char ToUpper(char);
int CopyFEAList(char, struct EAOP far *);
unsigned Init_SHUTDOWN_Data(char far *, char far *);
void Log_SD_Stats(struct SD_FS_stats far *);
int Getvpfsd(unsigned short hVPB);
int SaveFlags(unsigned short flags);

extern void int3();
extern void far memcop(char far * src, char far * dst, unsigned short count);

void PopupMsg(char far * Msg);

/*
* Global Data
* =====
*/

int _actused = 0; /* Get rid of the runtime */

char FS_NAME[] = FSDNAME;

```

```

char Version[] = VERSION;
char SignOnMsg[] = SIGNONMSG;

/* FEAsel is only initialized if some test which uses EA's is run */
5 unsigned short FEAsel; /* PUT A SEMAPHORE ON THIS LATER */
unsigned EAType = NICE_EAS; /* returned by FS_OPENCREATE for pfgenflag param */

/* OpenAction is returned as the action parameter from FS_OPENCREATE */
unsigned OpenAction = FILE_CREATED; /* default action */
10

/* pSDdata is only initialized for fsd234.exe -- testing FS_SHUTDOWN */
struct SD_stats far * pSDdata = LNULL;
unsigned Is_Shutdown_Test = INVALID; /* BEWARE! Still set after shutdown */
unsigned SD_Status = BEFORE_SD;
15

char Ddevfsd[sizeof(long)] = DEVFSD_MSG;
char BadData[sizeof(long)] = "bad";
char Dvpfsd[sizeof(struct vpfsd)] = VPFSD_MSG;

20 /*=====
=====*/
/**** PopupMsg - send message to user through FSH_CRITEROR
*
* ENTRY Msg - message string
25 * EXIT -none-
* RETURN -none-
*/
void PopupMsg(Msg)
char far *Msg;
30 {
int rc,MsgLen = 0;
char far *subs = "\0\0\0";

while(*(Msg+(MsgLen++)) != '\0'); /* get string length */
35 rc = FSH_CRITEROR(MsgLen,Msg,0,subs,CE_ALLABORT);
}

/*=====
=====*/
40 /**** FS_INIT - initialization entry
*
* ENTRY szParm - command line string from IFS= in config.sys
* DevHelp- address of the devhelp callgate
45 * pMiniFSD - null except when booting from a volume managed by
an FSD and the exported name of the FSD matches the exported
name of the mini-FSD. In this case, pMiniFSD will point to
data established by the mini-FSD (see mFS_INIT)
*
* EXIT -none-
50 * RETURN Error code= NO_ERROR
*
* WARNING:
55 *
* EFFECTS: prints sign on message to stdout at system initialization

```

```

* initializes data selector/semaphore array
*/

int far pascal FS_INIT(szParm, DevHelp, pMiniFSD)
5 char far *szParm;
 unsigned long DevHelp;
 unsigned long far *pMiniFSD;
 {
 unsigned short byteswritten;
10 int i;
 char ch;

 if (szParm != NULL)
 while (*szParm != '\0') {
15 switch (*szParm) {
 case 'D':
 int3();
 break;
 case 'd':
20 int3();
 break;
 }
 szParm++;
 }

25 DOSWRITE((unsigned short) 1,
 (char far *)SignOnMsg,
 (unsigned short)(sizeof(SignOnMsg)-1),
 (unsigned far *)&byteswritten);

30 DOSWRITE((unsigned short) 1,
 (char far *)Version,
 (unsigned short)(sizeof(Version)-1),
 (unsigned far *)&byteswritten);

35 Init_EnableFS_Tab(); /* initialize RAM semaphores for each FS_entry pt */

 return NO_ERROR;
40 }

/*=====
=====*/
/**** FS_FSCTL - exported routine for DOSFSCTL processing
45 *
 * ENTRY pArgdat - points to file system info structs psffsi,psffsd
 * iArgType - tells how to interpret pArgdat
 * func - function code
 * pParm - parameters if needed
 * lenParm - size of pParm data
50 * lenData - size of pData area
 * plenParmOut - size of pParm buffer sent back
 * plenDataOut - size of pData buffer sent back
 * EXIT pData - return results in this block
55 * RETURN Error code
 *

```

```

* This routine performs many different functions to provide an interface
* of communication between a ring 3 application and this test FSD.
*
* WARNING:
5 * This routine validates the pData buffer, it will be assumed
 * that that address remains valid throughout this FSD call.
 *
* EFFECTS:
 * If func so specifies, the whole selector table will be invalidated.
10 * This should only be used if a thread/process dies before using this
 * routine to invalidate its table entry. A reboot may be used instead.
 *
 */

15 int far pascal FS_FSCTL(pArgDat, iArgType, func, pParm, lenParm,
 plenParmOut, pData, lenData, plenDataOut)
 union argdat far *pArgDat;
 unsigned short iArgType;
 unsigned short func;
20 char far * pParm;
 unsigned short lenParm;
 unsigned short far * plenParmOut;
 char far * pData;
 unsigned short lenData;
25 unsigned short far * plenDataOut;
 {
 int rc, index, LName = 0;
 unsigned short sel, off, LSeg;
 FsctlRec far * pBuf;

30
 rc = FSH_PROBEBUF(WRITEPROBE, (char far *)plenDataOut, sizeof(short));
 if (rc) {
 PopupMsg("FS_FSCTL: bad plenDataOut pointer");
 return rc;
35
 }
 rc = FSH_PROBEBUF(WRITEPROBE, (char far
*)plenParmOut, sizeof(short));
 if (rc) {
 PopupMsg("FS_FSCTL: bad plenParmOut pointer");
40
 return rc;
 }
 *plenDataOut = *plenParmOut = 0;

 /* we are TESTING this entry point */
45 if (IsEnabled(ENT_FSCTL, (int far *)&index)) {

 rc = Disable(ENT_FSCTL, index);
 if (rc) return rc;

50
 /* allocate a GDT selector for returned data */
 LSeg = sizeof(FsctlRec);
 rc = InitTCBInstanceData((unsigned short far *)&sel,
 (unsigned short far *)&off,
 (int far *)&index, LSeg);
55
 if (rc || (index == -1)) goto abort;

```

```

/* make a far pointer to the buffer just allocated */
pBuf = (FscctlRec far *)MAKEFP(sel,0);

/* copy this entrypt's ID into buffer */
pBuf->owner = ENT_FSCTL;

/* copy the pArgDat data */
switch (iArgType) {
 case 1: /* FileHandle directed case */
 pBuf->thesffsi = *(pArgDat->sf.psffsi);
 pBuf->thesffsd = *(pArgDat->sf.psffsd);
 break;
 case 2: /* PathName directed case */
 pBuf->thecdfsi = *(pArgDat->cd.pcdfsi);
 pBuf->thecdfsd = *(pArgDat->cd.pcdfsd);

 /* get pName string length + null */
 while(*(pArgDat->cd.pPath+(LName+ ++)) != '\0');
 off = FIELDOFFSET(FscctlRec,pName);
 rc = CopyParam(sel,(unsigned short far *)&off,
 LName,pArgDat->cd.pPath);
 if (rc) goto abort;

 pBuf->iCurDirEnd = pArgDat->cd.iCurDirEnd;
 break;
 default: break;
}

pBuf->iArgType = iArgType;
pBuf->func = func;
pBuf->pParam = pParam;
pBuf->lenParam = lenParam;
pBuf->plenParamOut = plenParamOut;
pBuf->pData = pData;
pBuf->lenData = lenData;
pBuf->plenDataOut = plenDataOut;
}
else { /* we are USING this entry point for communication */

 switch (func) { /* note no break;'s */
 case FUNC_GET_BUF:
 return RetrieveBuf(pData,lenData);
 case FUNC_DEL_BUF:
 return DeleteTabEntry();
 case FUNC_ENABLE:
 rc = FSH_PROBEBUF(READPROBE,pParam,sizeof(int));
 if (rc) {
 PopupMsg((char far *)"FS_FSCTL: bad pParam");
 return rc;
 }
 /*pParam buffer contains the ENT x ID to enable*/
 return Enable(((int far *)pParam)[0]);
 case FUNC_DISABLE:
 rc = FSH_PROBEBUF(READPROBE,pParam,sizeof(int));
 if (rc) {
 PopupMsg((char far *)"FS_FSCTL: bad pParam");
 }
 }
}

```

```

 return rc;
 }
 /* pParm buffer contains ENT x ID to disable */
 if (IsEnabled(((int far *)pParm)[0],
 5 (int far *)&index)) {
 return Disable(((int far *)pParm)[0],index);
 }
 /* no error if not disabled to begin with */
 return NO_ERROR;
10 case FUNC_INIT_FEA_SEG:
 /*delete if can figure out how to alloc seg at init */
 return Init_FEA_Tab();
 case FUNC_DEL_FEALIST:
 rc =
15 FSH_PROBEBUF(READPROBE,pParm,sizeof(char));
 if (rc) {
 PopupMsg((char far *)"FS_FSCTL: bad pParm");
 return rc;
 }
 return DelFEAList(pParm[0]);
 case FUNC_SET_EATYPE:
 rc = FSH_PROBEBUF(READPROBE,pData,
20 sizeof(EAtype));
 if (rc) {
 PopupMsg((char far *)"FS_FSCTL: bad pData");
 return rc;
 }
 EAtype = *(unsigned far *)pData;
 return NO_ERROR;
25 case FUNC_SET_OPENACTION:
 rc = FSH_PROBEBUF(READPROBE,pData,
 sizeof(OpenAction));
 if (rc) {
 PopupMsg((char far *)"FS_FSCTL: bad pData");
 return rc;
 }
 OpenAction = *(unsigned far *)pData;
 return NO_ERROR;
30 case FUNC_SET_SHUTDOWN_DATA:
 return Init_SHUTDOWN_Data(pData,pParm);
 case GET_ERR_INFO: /* return error code info */
 return NO_ERROR;
 case GET_MIN_MAX_EA: /* return max/min EA
35 sizes */
 return NO_ERROR;
 case FUNC_CLEAR_TABLE: /* clear selector table */
 ClearTable();
 return NO_ERROR;
 case FUNC_SEL_TAB_DUMP:
 return SelTabDump(pData,lenData);
50 case FUNC_TCB_BUF_COUNT:
 return GetBufCount(pData,lenData);
 default:
 return FSCTL_UNKNOWN_FUNCTION;
55 } /* end switch */
}

```

```

 return NO_ERROR;
abort:
 DeleteTabEntry();
 return rc;
} /* FS_FSCTL() */

/*=====
=====*/
/** FS_ATTACH - attach a remote drive or pseudo character device
 *
 * ENTRY flag - 0=attach;1=detach;2=query
 * pDev - pointer to text of drive: or \dev\device
 * pvpsfd - pointer to vpsfd structure
 * pcdfsd - pointer to current directory info
 * pParm - address of application parameter area
 * pLen - pointer to length of application parameter area
 *
 * EXIT -none-
 * RETURN NO_ERROR if successful attach/detach; error code if attach 'fails'
 *
 * If enabled:
 * This routine obtains an entry in the thread/GDT selector table, and
 * records whether the 3rd parameter is null or not. When attaching a
 * remote pseudo-character device, this parameter should be null. It should
 * be valid for all other instances.
 *
 * If a device is attached, the cdfsd (treated as a DWORD devfsd for devices)
 * is filled in. This same data should arrive intact to FS_OPENCREATE when
 * the device is opened. If a drive is attached, the vpsfd is filled in.
 * This same data should arrive intact to any FS_ entry receiving an hVPB
 * referring to this drive. The cdfsd is also filled in, but with some 'bad'
 * data, just in case the cdfsd data is passed to OPENCREATE instead of the
 * devfsd data, when a device is opened.
 *
 * WARNING:
 *
 * EFFECTS:
 * If we are not testing this entrypoint explicitly, and if detaching a
 * drive, any EA's set for this drive are deleted.
 */
int far pascal
FS_ATTACH(flag, pDev, pvpsfd, pcdfsd, pParm, pLen)
 unsigned short flag;
 char far * pDev;
 char far * pvpsfd;
 char far * pcdfsd;
 char far * pParm;
 unsigned short far * pLen;
{
 int rc,index;
 unsigned short sel, off, Valid, LSeg=0;

```

```

if (IsEnabled(ENT_ATTACH,(int far *)&index)) {

 rc = Disable(ENT_ATTACH,index);
 if (rc) return rc;

5 if (flag == 0) { /* attach */
 LSeg = sizeof(short) * 2; /* save length & valid flag */
 rc = InitTCBInstanceData((unsigned short far *)&sel,
 (unsigned short far *)&off,
 (int far *)&index,LSeg);
10 if (rc || (index == -1)) goto abort;

 if (pvpfsd == LNULL) { /* attaching device */
 Valid = INVALID;
 /* fill in cdfsd (treated as devfsd for a device) */
 memcpy((char far *)Ddevfsd,(char far *)pcdfsd,
15 sizeof(long));

 }
 else { /* attaching drive */
 Valid = VALID;
 /* fill in the vpfsd for drive */
 memcpy((char far *)Dvpfsd,(char far *)pvpfsd,
20 sizeof(struct vpfsd));

 /*
 * fill in cdfsd with "bad" in case it gets sent to
 * opencreate when opening a device instead of being
 * sent 'devfsd'
 */
 memcpy((char far *)BadData,(char far *)pcdfsd,
25 sizeof(long));

 }

 /* record whether the 3rd parameter is valid or not */
 rc = CopyParam(sel,(unsigned short far *)&off,
30 sizeof(short),(char far *)&Valid);
 if (rc) goto abort;

 }
 return NO_ERROR; /* no error for detach or query */
40 }

 /* if detaching a DRIVE, delete (if) any EA's corresponding to drive */
 if ((flag == FSA_DETACH) && (pvpfsd != LNULL)) {
 DelFEAList(pDev[0]); /* don't worry if error */
45 }

 return NO_ERROR;

abort:
 DeleteTabEntry();
50 return rc;
}

/*=====
=====*/
/** FS_OPENCREATE -- open/create file entry point
55 *
 * FS_OPENCREATE(pcdfsi, pcdfsd, pName, iCurDirEnd, psffsi, psffsd,

```



```

* fhandflag, openflag, pAction, attr, pEABuf)
*
* ENTRY - pcdfsi -> file system independent working directory struct
* pcdfsd -> file system dependent working directory struct
5 * pName -> asciiz name of file
* iCurDirEnd = index of the end of the current directory in pName
* psffsi -> file system independent portion of file instance
* psffsd -> file system dependent portion of file instance
* fhandflag = desired sharing mode and access mode
* openflag = action taken when the file is present or absent
10 * pAction -> action taken variable
* attr = OS/2 file attributes
* pEABuf -> extended attribute buffer
* EXIT - returns NO_ERROR because this is a FAKE OpenCreate.
15 *
* Named pipe considerations:
* THIS CALL DOES NOT DO AN OPEN. It is merely a means of
* opening a named pipe without making or connecting it first
*
20 * If enabled:
* This routine copies its first 2 parameters to a TCB instance data area
* for verification
*
* In general:
25 * EAs may be set on a per-drive basis -- i.e. all files pertaining
* to a particular drive will have the last EA list set for any file on
* the same drive (if any). So, care must be taken if setting different
* EA lists for files on the same drive -- be sure the EA's you set are
* not cancelled out by a subsequent DOSOPEN2's.
30 * sffsi and sffsd are filled in with some bogus values. The action
* returned is always file-created.
*/
int far pascal
FS_OPENCREATE(pcdfsi, pcdfsd, pName, iCurDirEnd, psffsi, psffsd, fhandflag,
35 openflag, pAction, attr, pEABuf, pfgenflag)
 struct cdfs far * pcdfsi;
 char far * pcdfsd;
 char far * pName;
 unsigned short iCurDirEnd;
40 struct sffsi far * psffsi;
 struct sffsd far * psffsd;
 unsigned short fhandflag;
 unsigned short openflag;
 unsigned short far * pAction;
45 unsigned short attr;
 char far * pEABuf;
 unsigned short far * pfgenflag;
{
 int rc, index, fsdrc = NO_ERROR;
50 unsigned short FEAoff, sel, off, Valid, LSeg=0;
 struct FEAList far *temp;

 if (!IsEnabled(ENT_OPENCREATE,(int far *)&index)) {

55 rc = Disable(ENT_OPENCREATE,index);
 if (rc) return rc;

```

```


 5 /* get the length of the GDT segment needed */
 LSeg = sizeof(GenericRec);
 rc = InitTCBInstanceData((unsigned short far *)&sel,
 (unsigned short far *)&off,
 (int far *)&index, LSeg);
 if (rc || (index == -1)) goto abort;

 10 /* record whether the 1st parameter is valid or not */
 Valid = (pcdfsi == LNULL) ? INVALID : VALID;
 rc = CopyParam(sel, (unsigned short far *)&off,
 sizeof(short), (char far *)&Valid);
 if (rc) goto abort;

 15 /* copy the contents of the devfsd field (cdfsd parameter) */
 rc = CopyParam(sel, (unsigned short far *)&off,
 sizeof(long), (char far *)&pcdfsd);
 if (rc) goto abort;
 20 }

 pAction = OpenAction; / action set via FSCTL */
 /* if we set action to undefined, then we want to explicitly fail */
 if (OpenAction == UNDEFINED_ACTION) fsdrc = ERROR_OPEN_FAILED;

 25 *pfgenflag = EAtype; /* this can be set via FSCTL */

 /* is this a file name? */
 if (pName[1] == ':') {
 30 psffsi->sfi_type = STYPE_FILE;
 /*
 * fill in the first byte of sffsd with current drive letter
 * if this is a file -- cheap way to let FS_FILEINFO know what
 * drive a file refers to. Assuming the next file queried is
 35 * the one opened here...
 */
 psffsd->sfd_work[0] = pcdfsi->cdi_curdir[0];
 psffsd->sfd_work[1] = '\0';

 40 /*
 * Set extended attributes --
 * if DOSOPEN2 sends in 0L for pEABuf, the kernel sets the
 * RPL bits in the SELECTOR to reflect ring3 protection level;
 * so mask off bits 0&1 of the selector before checking for
 45 * NULL
 */
 if ((unsigned long)pEABuf & (~0x00030000)) { /* pEABuf not null */
 rc = FSH_PROBEBUF(READPROBE, pEABuf, sizeof(struct
 50 EAOP));

 if (rc) {
 PopupMsg((char far *)"FS_OPEN:bad pEABuf");
 return rc;
 }
 }
 /* calculate offset into the FSD's per/drive FEA table */
 55 FEAOFF = CalcFEAOFF(pName[0]);
 temp = ((struct EAOP far *)pEABuf)->fpFEAList;

```

```

 if ((unsigned short)(temp->feal_cbList) > MAXFEACBLIST) {
 fsdrc = FSD_FEALIST_TOO_LONG;
 }
 else {
5 CopyParam(FEAsel, (unsigned short far *)&FEAoff,
 (unsigned short)(temp->feal_cbList),
 (char far *)temp);
 }
 }
10
 /* is it a device? */
 else if ((pName[1] == 'D') && (pName[2] == 'E') && (pName[3] == 'V')) {
 psffsi->sfi_type = STYPE_DEVICE;
15 }
 /* must be a named pipe */
 else {
 psffsi->sfi_type = STYPE_NMPIPE;
 /* fill in first byte of sffsd */
20 psffsd->sfd_work[0] = BOGUS_WORK_CHAR;
 psffsd->sfd_work[1] = '\0';
 }

 /* fill in some fields of the sffsi with a bogus value */
25 psffsi->sfi_ctime = BOGUS_CTIME;
 psffsi->sfi_cdate = BOGUS_CDATE;
 psffsi->sfi_atime = BOGUS_ETIME;
 psffsi->sfi_adate = BOGUS_ATE;
 psffsi->sfi_mtime = BOGUS_MTIME;
30 psffsi->sfi_mdate = BOGUS_MDATE;
 psffsi->sfi_size = (long)BOGUS_SIZE;
 psffsi->sfi_position = (long)BOGUS_POS;

 if (Is_Shutdown_Test) Log_SD_Stats(&(pSDdata->Open_stats));
35
 return fsdrc;

abort:
 DeleteTabEntry();
40 return rc;
}

/*=====
=====*/
45 /*** FS_FILEINFO - returns information for a specific file
 *
 * ENTRY flag - indicates retrieval vs. setting of information
 * psffsi - pointer to file system independent data
 * psffsd - pointer to file system dependent data
50 * level - information level to be returned
 * pData - address of application data area
 * cbData - length of the application data area
 * IOflag - per handle flag
 *
 * EXIT -none-
55 * RETURN NO_ERROR if successful
 * If enabled:

```

```

*
* WARNING:
* EFFECTS: EA's will always be returned for query of level 4
*/
5 int far pascal
 FS_FILEINFO(flag, psffsi, psffsd, level, pData, cbData, IOflag)
 unsigned short flag;
 struct sffsi far * psffsi;
 struct sffsd far * psffsd;
10 unsigned short level;
 char far * pData;
 unsigned short cbData;
 unsigned short IOflag;
 {
15 int rc, index;
 unsigned short sel, off, LSeg;
 FileInfoRec far * pBuf;

 if (IsEnabled(ENT_FILEINFO,(int far *)&index)) {
20 rc = Disable(ENT_FILEINFO,index);
 if (rc) return rc;

 /* allocate a GDT selector for returned data */
 LSeg = sizeof(FileInfoRec);
25 rc = InitTCBInstanceData((unsigned short far *)&sel,
 (unsigned short far *)&off,
 (int far *)&index,LSeg);
 if (rc || (index == -1)) goto abort;

30 pBuf = (FileInfoRec far *)MAKEFP(sel,0);

 pBuf->owner = ENT_FILEINFO;
 pBuf->flag = flag;
 pBuf->thesffsi = *psffsi;
35 pBuf->thesffsd = *psffsd;
 pBuf->level = level;
 pBuf->pData = pData;
 /* cbData should be cbList (of FEA list) + sizeof EAOP */
 pBuf->cbData = cbData;
40 pBuf->IOflag = IOflag;
 }
 /* always copy the FEA list for retrieval of level 4 info */
 if ((flag == 0) && (level == 4)) {
 /*if sfd_work is trashed--use FSH_NAMEFROMSFN with LDT buf*/
45 return CopyFEAList(psffsd->sfd_work[0],
 (struct EAOP far *)pData);
 }
 else return NO_ERROR;

50 abort:
 DeleteTabEntry();
 return rc;
 }
55 /*=====
 =====*/

```

```

 /*** FS_PATHINFO - get/set a file's informations
 *
 * ENTRY flag - retrieve=0; set=1; all other values reserved
 *
 * pcdfsi - ptr to file system independent curdir data
 5 * pcdfsd - ptr to file system dependend curdir data
 * pName - ptr to name of file or directory
 * iCurDirEnd - index of the end of the curdir in pName
 * level - level of info to return in pData
 * pData - ptr to application data area
 10 * cbData - length of application data area
 * EXIT pData - filled with requested info (if flag = 0)
 * RETURN
 * WARNING
 * EFFECTS:
 15 * EA's will always be returned for query of level 4
 *
 *
 */
 20
 int far pascal
 FS_PATHINFO(flag, pcdfsi, pcdfsd, pName, iCurDirEnd, level, pData, cbData)
 unsigned short flag;
 struct cdfs far * pcdfsi;
 25 struct cdfs far * pcdfsd;
 char far * pName;
 unsigned short iCurDirEnd;
 unsigned short level;
 char far * pData;
 30 unsigned short cbData;
 {

 int rc, index;
 unsigned short sel, off, LSeg, LName = 0;
 35 PathInfoRec far * pBuf;

 if (!IsEnabled(ENT_PATHINFO,(int far *)&index)) {
 rc = Disable(ENT_PATHINFO,index);
 if (rc) return rc;
 40
 /* allocate a GDT selector for returned data */
 LSeg = sizeof(PathInfoRec);
 rc = InitTCBInstanceData((unsigned short far *)&sel,
 (unsigned short far *)&off,
 45 (int far *)&index,LSeg);
 if (rc || (index == -1)) goto abort;

 pBuf = (PathInfoRec far *)MAKEFP(sel,0);

 50 pBuf->owner = ENT_PATHINFO;
 pBuf->flag = flag;
 pBuf->thecdfsi = *pcdfs;
 pBuf->thcdfsd = *pcdfs;

 55 /* copy the pName parameter */
 off = FIELDOFFSET(PathInfoRec,pName);

```

```

while(*(pName+(LName++)) != '\0'); /* get the name length */
rc = CopyParam(sel,(unsigned short far *)&off,
 LName,pName);
if (rc) goto abort;

5 pBuf->iCurDirEnd = iCurDirEnd;
 pBuf->level = level;
 pBuf->pData = pData;
 pBuf->cbData = cbData;

10 }

 /* always copy the FEA list for retrieval of level 4 info */
 if ((flag == 0) && (level == 4)) {
15 return CopyFEAList(pName[0],(struct EAOP far *)pData);
 }
 else return NO_ERROR;

abort:
20 DeleteTabEntry();
 return rc;

}

/*=====
25 =====*/
int far pascal
FS_SHUTDOWN(type,reserved)
unsigned short type;
unsigned long reserved;
30 {

 #if defined(generic)

 /*
35 * If reserved is not 0, it will be recorded in the pSDdata segment.
 * If it is bad on > 1 call, the last bad value received will be
 * reflected when the data is retrieved by the ring3 test.
 */
 if (reserved != 0L) {
40 pSDdata->General_stats.FS_SD_bad_reserved = reserved;
 }

 switch (type) {
 case SHUTDOWN_START:{
45 SD_Status = DURING_SD;
 /* signal the ring3 test's worker threads to begin */
 if (pSDdata != LNULL) {
 FSH_SEMCLR((char far *)pSDdata-
>General_stats.Signal_SD_RAM_sem);
50 }
 break;
 }
 case SHUTDOWN_END: {
55 SD_Status = AFTER_SD;
 break;
 }
 }

```

```

 default: {
 /* the last bad type param received will be recorded */
 pSDdata->General_stats.FS_SD_bad_type = type;
 }
5 } /* switch */

 /* record the thread which called FS_SHUTDOWN */
 if (pSDdata != LNULL) { /* only if the test is calling us.. */
 Log_SD_Stats(&(pSDdata->Shutdown_stats));
10 /* allow other threads to run */
 FSH_YIELD();
 }
 /* even if a param is bad, return 0; error will be recorded in GDT */

15 #endif

 return NO_ERROR;
 }
 /*=====
=====*/
20 /* return the uppercase of an alphabetic character, else return 0 */
 char ToUpper(letter)
 char letter;
 {
 if (ISBETWEEN('A','Z',letter)) return letter;
25 else if (ISBETWEEN('a','z',letter)) return (letter - 'a' + 'A');
 else return 0x00;
 }

 /*=====
=====*/
30 /**** Init_FEA_Tab() -- allocate a GDT for the FSD's per/drive FEA lists
 *
 * ENTRY none
 * EXIT
35 * RETURN: appropriate error code
 *
 * WARNING:
 * If multiple threads (processes) will be accessing the same FSD built
 * from this file, a semaphore must be added to the section of code as
40 * indicated. We only want the GDT selector to be allocated once.
 *
 * EFFECTS:
 * The FSD will allocate itself a GDT segment to use for storing FEA lists
 * on a per/drive (not per/file) basis. The GDT block is divided into
45 * NUMDRIVE sub-blocks. Each drive has MAXFEACBLIST bytes
 * in which to store its FEA list. Initially, each FEA list is empty. This
 * is indicated by storing the sizeof the cbList field of an FEA list in the
 * first bytes of each sub-block. This GDT segment is only allocated if
 * a test application explicitly instructs it (via FS_FSCTL).
50 */
 /* long size is hardcoded here for cblist */
 int Init_FEA_Tab()
 {
 char far * FEATab;
55 int i = 0, rc;
 unsigned FEAoff;

```

```

/*=====HEY!!! GET A SEMAPHORE=====*/
if (FEAset != 0) return FEA_SEGMENT_EXISTS;
/* allocate a ring 0, GDT, nonswappable segment selector */
rc = FSH_SEGALLOC(0x0000, (long)(MAXFEACBLIST*NUMDRIVES),
 (unsigned short far *)&FEAset);
/*=====HEY!!! RELEASE THE SEMAPHORE=====*/
if (rc) return rc;
for (i='A'; i<='Z'; i++) {
 FEAoff = CalcFEAoff(i);
 FEATab = (char far *)MAKEFP(FEAset, FEAoff);
 ((unsigned long far *)FEATab)[0] = (long)NULLFEACBLIST;
}
return NO_ERROR;
}
/*=====
=====*/
/**DelFEAList -- mark the FEA list for a drive as empty
*
* ENTRY drive - which drive to mark an empty FEA list for
* EXIT
* RETURN appropriate error code
* WARNING:
* EFFECTS:
* The first bytes of the appropriate sub-block of the FSD's FEA segment
* is marked with the sizeof the cbList field of an FEAList. This indicates
* that the list is empty, i.e. only is long enough to hold the length of
* the field itself, which indicates the length of the list.
*
*/
int DelFEAList(drive)
char drive;
{
 char far * FEATab, updrive;

 if (!(updrive = ToUpper(drive))) return ERROR_INVALID_DRIVE;
 if (FEAset != 0) {
 FEATab = (char far *)MAKEFP(FEAset, CalcFEAoff(updrive));
 ((unsigned long far *)FEATab)[0] = (long)NULLFEACBLIST;
 return NO_ERROR;
 }
 else return NULL_FEA_SELECTOR;
}
/*=====
=====*/
/**CopyFEAList -- copy the FEA sub-block for this drive to an FEA buffer
*
* ENTRY drive - which drive should we get the FEA list for?
* pEAOP - pointer to an EAOP list which holds the pointer
* to the FEA buffer we will copy our FEA list to
* EXIT
* RETURN
* WARNING: FEA lists are limited to 64k in OS/2 version 1.2
* EFFECTS:
*
* An EAOP structure contains a pointer to an FEA list buffer. It is

```



```

/* === HEY!!! GET A SEMAPHORE === */
if (FEAset != 0) return FEA_SEGMENT_EXISTS;
/* allocate a ring 0, GDT, nonswappable segment selector */
rc = FSH_SEGALLOC(0x0000, (long)(MAXFEACBLIST*NUMDRIVES),
 (unsigned short far *)&FEAset);
/* === HEY!!! RELEASE THE SEMAPHORE === */
if (rc) return rc;
for (i='A'; i<='Z'; i++) {
 FEAoff = CalcFEAoff(i);
 FEATab = (char far *)MAKEFP(FEAset, FEAoff);
 ((unsigned long far *)FEATab)[0] = (long)NULLFEACBLIST;
}
return NO_ERROR;
}

/*=====
=====*/
/** DelFEAList -- mark the FEA list for a drive as empty
*
* ENTRY drive - which drive to mark an empty FEA list for
* EXIT
* RETURN appropriate error code
* WARNING:
* EFFECTS:
* The first bytes of the appropriate sub-block of the FSD's FEA segment
* is marked with the sizeof the cbList field of an FEAList. This indicates
* that the list is empty, i.e. only is long enough to hold the length of
* the field itself, which indicates the length of the list.
*
*/
int DelFEAList(drive)
char drive;
{
 char far * FEATab, updrive;

 if (!(updrive = ToUpper(drive))) return ERROR_INVALID_DRIVE;
 if (FEAset != 0) {
 FEATab = (char far *)MAKEFP(FEAset, CalcFEAoff(updrive));
 ((unsigned long far *)FEATab)[0] = (long)NULLFEACBLIST;
 return NO_ERROR;
 }
 else return NULL_FEA_SELECTOR;
}

/*=====
=====*/
/** CopyFEAList -- copy the FEA sub-block for this drive to an FEA buffer
*
* ENTRY drive - which drive should we get the FEA list for?
* pEAOP - pointer to an EAOP list which holds the pointer
* to the FEA buffer we will copy our FEA list to
* EXIT
* RETURN
* WARNING: FEA lists are limited to 64k in OS/2 version 1.2
* EFFECTS:
*
* An EAOP structure contains a pointer to an FEA list buffer. It is

```

```

 * this pointer that will be used as the destination of the FEA list copy.
 * The appropriate source FEA list will be calculated from 'drive' and
 * the FSD's FEA segment.
 *
5 */
 int CopyFEAList(drive,pEAOP)
 char drive;
 struct EAOP far * pEAOP;
10 {
 int rc;
 struct FEAList far * dest;
 char far * source;
 unsigned count;

15 rc = FSH_PROBEBUF(READPROBE,(char far *)pEAOP,sizeof(struct
 EAOP));
 if (rc) {
 PopupMsg((char far *)"CopyFEAList: bad pEAOP");
 return rc;
20 }

 dest = pEAOP->fpFEAList;
 /* note conversions from long to short for version 1.2 */
 rc = FSH_PROBEBUF(WRITEPROBE,(char far *)dest,
25 (unsigned short)(dest->feal_cbList));
 if (rc) {
 PopupMsg((char far *)"CopyFEAList: bad fpFEAList");
 return rc;
30 }

 source = (char far *)MAKEFP(FEAsel,CalcFEAoff(drive));
 count = (unsigned short)((((unsigned long far *)source)[0]));

 /* EA lists are restricted to 64k in 1.2 */
35 if (((unsigned short)(dest->feal_cbList) < count) {
 return ERROR_BUFFER_OVERFLOW;
 }

 memcop(source,(char far *)dest,count);
40 return NO_ERROR;
 }

 unsigned Init_SHUTDOWN_Data(pGDT_sel,pCBsel)
 char far * pGDT_sel;
45 char far * pCBsel;
 {
 unsigned rc;

 rc = FSH_PROBEBUF(WRITEPROBE,pGDT_sel,sizeof(short));
50 if (rc) {
 PopupMsg((char far *)"Init_SHUTDOWN_Data:bad pGDT_sel");
 return rc;
 }

 rc = FSH_PROBEBUF(READPROBE,pCBsel,sizeof(short));
55 if (rc) {
 PopupMsg((char far *)"Init_SHUTDOWN_Data:bad pCBsel");

```

```

 pBuf = (CommitnCloseRec far *)MAKEFP(sel,0);

 pBuf->owner = ENT_COMMIT;
 pBuf->type = type;
5 pBuf->IOflag = IOflag;
 }

 if (Is_Shutdown_Test) Log_SD_Stats(&(pSDdata->Commit_stats));
 return NO_ERROR;
10 abort:

 DeleteTabEntry();
 return rc;
 }

15 int far pascal
 FS_READ(psffsi, psffsd, pData, pLen, IOflag)
 struct sffsi far * psffsi;
 struct sffsd far * psffsd;
 char far * pData;
20 unsigned short far * pLen;
 unsigned short IOflag;
 {
 int rc, index;

25 if (IsEnabled(ENT_READ,(int far *)&index)) {
 rc = Disable(ENT_READ,index);
 if (rc) return rc;
 return SaveFlags(IOflag);
 }

30 if (Is_Shutdown_Test) Log_SD_Stats(&(pSDdata->Read_stats));
 return NO_ERROR;
 }

35 int far pascal
 FS_WRITE(psffsi, psffsd, pData, pLen, IOflag)
 struct sffsi far * psffsi;
 struct sffsd far * psffsd;
 char far * pData;
40 unsigned short far * pLen;
 unsigned short IOflag;
 {
 int rc, index;

45 if (IsEnabled(ENT_WRITE,(int far *)&index)) {
 rc = Disable(ENT_WRITE,index);
 if (rc) return rc;
 return SaveFlags(IOflag);
 }

50 if (Is_Shutdown_Test) Log_SD_Stats(&(pSDdata->Write_stats));
 return NO_ERROR;
 }

55 int far pascal
 FS_DELETE(pcdfsi, pcdfsd, pFile, iCurDirEnd)

```

```

 struct cdfsi far * pcdfsi;
 struct cdfsd far * pcdfsd;
 char far * pFile;
 unsigned short iCurDirEnd;
5 {
 if (Is_Shutdown_Test) Log_SD_Stats(&(pSDdata->Delete_stats));
 return NO_ERROR;
 }

10 int far pascal
 FS_FLUSHBUF(hVPB, flag)
 unsigned short hVPB;
 unsigned short flag;
 {
15 int rc, index;

 if (IsEnabled(ENT_FLUSHBUF,(int far *)&index)) {
 rc = Disable(ENT_FLUSHBUF,index);
 if (rc) return rc;
20 return Getvpfsd(hVPB);
 }

 if (Is_Shutdown_Test) Log_SD_Stats(&(pSDdata->Flushbuf_stats));
 return NO_ERROR;
25 }

 int far pascal
 FS_CHDIR(flag, pcdfsi, pcdfsd, pDir, iCurDirEnd)
 unsigned short flag;
30 struct cdfsi far * pcdfsi;
 struct cdfsd far * pcdfsd;
 char far * pDir;
 unsigned short iCurDirEnd;
 {
35 if (Is_Shutdown_Test) Log_SD_Stats(&(pSDdata->Other_stats));
 return NO_ERROR;
 }

 int far pascal
40 FS_CHGFILEPTR(psffsi, psffsd, offset, type, IOflag)
 struct sffsi far * psffsi;
 struct sffsd far * psffsd;
 long offset;
 unsigned short type;
45 unsigned short IOflag;
 {
 int rc, index;

 if (IsEnabled(ENT_CHGFILEPTR,(int far *)&index)) {
50 rc = Disable(ENT_CHGFILEPTR,index);
 if (rc) return rc;
 return SaveFlags(IOflag);
 }

55 if (Is_Shutdown_Test) Log_SD_Stats(&(pSDdata->Other_stats));
 return NO_ERROR;

```

```

}
```

```

int far pascal
```

```

FS_CLOSE(type, IOflag, psffsi, psffsd)
```

```

 unsigned short type;
```

```

 unsigned short IOflag;
```

```

 struct sffsi far * psffsi;
```

```

 struct sffsd far * psffsd;
```

```

{
```

```

 int rc, index;
```

```

 unsigned short sel, off, LSeg;
```

```

 CommitnCloseRec far * pBuf;
```

```

 if (IsEnabled(ENT_CLOSE,(int far *)&index)) {
```

```

 rc = Disable(ENT_CLOSE,index);
```

```

 if (rc) return rc;
```

```

 LSeg = sizeof(CommitnCloseRec);
```

```

 rc = InitTCBInstanceData((unsigned short far *)&sel,
```

```

 (unsigned short far *)&off,
```

```

 (int far *)&index,LSeg);
```

```

 if (rc || (index == -1)) goto abort;
```

```

 pBuf = (CommitnCloseRec far *)MAKEFP(sel,0);
```

```

 pBuf->owner = ENT_CLOSE;
```

```

 pBuf->type = type;
```

```

 pBuf->IOflag = IOflag;
```

```

 }
```

```

 if (Is_Shutdown_Test) Log_SD_Stats(&(pSDdata->Other_stats));
```

```

 return NO_ERROR;
```

```

abort:
```

```

 DeleteTabEntry();
```

```

 return rc;
```

```

}
```

```

int far pascal
```

```

FS_COPY(mode, pcdfsd, pcdfsi, pSrc, iSrcCurDirEnd, pDst, iDstCurDirEnd, flags)
```

```

 unsigned short mode;
```

```

 struct cdfsi far * pcdfsd;
```

```

 struct cdfsd far * pcdfsi;
```

```

 char far * pSrc;
```

```

 unsigned short iSrcCurDirEnd;
```

```

 char far * pDst;
```

```

 unsigned short iDstCurDirEnd;
```

```

 unsigned short flags;
```

```

{
```

```

 if (Is_Shutdown_Test) Log_SD_Stats(&(pSDdata->Other_stats));
```

```

 return(ERROR_CANNOT_COPY);
```

```

}
```

```

void far pascal
```

```

FS_EXIT(uid, pid, pdb)
```

```

 unsigned short uid;
```

```

 unsigned short pid;
```

```

 unsigned short pdb;
 {
 if (Is_Shutdown_Test) Log_SD_Stats(&(pSDdata->Other_stats));
5 }

 int far pascal
 FS_FILEATTRIBUTE(flag, pcdfsi, pcdfsd, pName, iCurDirEnd, pAttr)
 unsigned short flag;
10 struct cdfsi far * pcdfsi;
 struct cdfsd far * pcdfsd;
 char far * pName;
 unsigned short iCurDirEnd;
 unsigned short far * pAttr;
15 {
 if (Is_Shutdown_Test) Log_SD_Stats(&(pSDdata->Other_stats));
 return ERROR_NOT_SUPPORTED;
 }

20 int far pascal
 FS_FILEIO(psffsi, psffsd, pCmdList, pCmdLen, poError, IOflag)
 struct sffsi far * psffsi;
 struct sffsd far * psffsd;
 char far * pCmdList;
25 unsigned short pCmdLen;
 unsigned short far * poError;
 unsigned short IOflag;
 {
 int rc, index;
30
 if (IsEnabled(ENT_FILEIO,(int far *)&index)) {
 rc = Disable(ENT_FILEIO,index);
 if (rc) return rc;
 return SaveFlags(IOflag);
35 }

 if (Is_Shutdown_Test) Log_SD_Stats(&(pSDdata->Other_stats));
 return ERROR_NOT_SUPPORTED;
 }

40 int far pascal
 FS_FINDCLOSE(pfsfsi,pfsfsd)
 struct fsfsi far * pfsfsi;
 struct fsfsd far * pfsfsd;
45 {
 if (Is_Shutdown_Test) Log_SD_Stats(&(pSDdata->Other_stats));
 return ERROR_NOT_SUPPORTED;
 }

50 int far pascal
 FS_FINDFROMNAME(pfsfsi, pfsfsd, pData, cbData, pcMatch, level, position,
 pName,flags)
 struct fsfsi far * pfsfsi;
55 struct fsfsd far * pfsfsd;
 char far * pData;

```

```

 unsigned short cbData;
 unsigned short far * pcMatch;
 unsigned short level;
 unsigned long position;
5 char far * pName;
 unsigned short flags;
 {
 if (Is_Shutdown_Test) Log_SD_Stats(&(pSDdata->Other_stats));
 return ERROR_NOT_SUPPORTED;
10 }

 int far pascal
 FS_FINDFIRST(pcdfsi, pcdfsd, pName, iCurDirEnd, attr, pfsfsi, pfsfsd, pData, cbData,
 pcMatch, level, flags)
15 struct cdfsi far * pcdfsi;
 struct cdfsd far * pcdfsd;
 char far * pName;
 unsigned short iCurDirEnd;
 unsigned short attr;
20 struct fsfsi far * pfsfsi;
 struct fsfsd far * pfsfsd;
 char far * pData;
 unsigned short cbData;
 unsigned short far * pcMatch;
25 unsigned short level;
 unsigned short flags;
 {
 if (Is_Shutdown_Test) Log_SD_Stats(&(pSDdata->Other_stats));
 return ERROR_NOT_SUPPORTED;
30 }

 int far pascal
 FS_FINDNEXT(pfsfsi, pfsfsd, pData, cbData, pcMatch, level, flag)
35 struct fsfsi far * pfsfsi;
 struct fsfsd far * pfsfsd;
 char far * pData;
 unsigned short cbData;
 unsigned short far * pcMatch;
 unsigned short level;
40 unsigned short flag;
 {
 if (Is_Shutdown_Test) Log_SD_Stats(&(pSDdata->Other_stats));
 return ERROR_NOT_SUPPORTED;
 }
45

 int far pascal
 FS_FSINFO(flag, hVPB, pData, cbData, level)
 unsigned short flag;
 unsigned short hVPB;
50 char far * pData;
 unsigned short cbData;
 unsigned short level;
 {
 int rc, index;
55
 if (IsEnabled(ENT_FSINFO,(int far *)&index)) {

```

```

 rc = Disable(ENT_FSINFO,index);
 if (rc) return rc;
 return Getvpfsd(hVPB);
 }
5
 if (Is_Shutdown_Test) Log_SD_Stats(&(pSDdata->Other_stats));
 return ERROR_NOT_SUPPORTED;
}

10 int far pascal
 FS_IOCTL(psfsi, psffsd, cat, func, pParm, lenParm, pData, lenData)
 struct sfsi far * psfsi;
 struct sffsd far * psffsd;
 unsigned short cat;
15 unsigned short func;
 char far * pParm;
 unsigned short lenParm;
 char far * pData;
 unsigned short lenData;
20 {
 return ERROR_NOT_SUPPORTED;
 }

 int far pascal
25 MFS_CHGFILEPTR(
 unsigned long offset,
 unsigned short type
)
 {
30 return ERROR_NOT_SUPPORTED;
 }

 int far pascal
 MFS_CLOSE()
35 {
 return ERROR_NOT_SUPPORTED;
 }

 int far pascal
40 MFS_INIT(
 void far * bootdata,
 char far * number_io,
 long far * vectorripl,
 char far * bpb,
45 unsigned long far * pMiniFSD,
 unsigned long far * dumpaddr
)
 {
 return ERROR_NOT_SUPPORTED;
50 }

 int far pascal
 MFS_OPEN(
 char far * name,
55 unsigned long far * size
)

```



```

 {
 return ERROR_NOT_SUPPORTED;
 }

5 int far pascal
 MFS_READ(
 char far * data,
 unsigned short far * length
)
10 {
 return ERROR_NOT_SUPPORTED;
 }

 int far pascal
15 MFS_TERM()
 {
 return ERROR_NOT_SUPPORTED;
 }

20 int far pascal
 FS_MKDIR(pcdfsi, pcdfsd, pName, iCurDirEnd, pEABuf, flags)
 struct cdfsi far * pcdfsi;
 struct cdfsd far * pcdfsd;
 char far * pName;
25 unsigned short iCurDirEnd;
 char far * pEABuf;
 unsigned short flags;
 {
30 if (Is_Shutdown_Test) Log_SD_Stats(&(pSDdata->Other_stats));
 return NO_ERROR;
 }

/* FS_MOUNT flags */
35 #define MOUNT_NEW 0
 #define MOUNT_REMOVED 1
 #define MOUNT_DISMOUNT 2
 #define MOUNT_FORMAT 3
40

/** FS_MOUNT - handle volume mounting
 *
 *
 * ENTRY flag - operation type
45 * pvpfsi - VPB info (file system independent)
 * pvpfsd - VPB info (file system dependent)
 * hVPB - handle to VPB
 * pBoot - boot sector contents
 * EXIT -none-
50 * RETURN NO_ERROR if match; !NO_ERROR if fails to match
 *
 * This routine will accept the volume if:
 * 1. The Boot_Sig field is 41d or greater
 * 2. The Boot_System_ID matches FS_NAME for all FS_NAME_LEN chars
55 *
 * WARNING:

```

```

*
* EFFECTS:
* uses FS_NAME[] for ID matching
*
5 */

int far pascal FS_MOUNT(flag, pvpfsi, pvpfsd, hVPB, pBoot)
unsigned short flag;
struct vpfsi far * pvpfsi;
10 struct vpfsd far * pvpfsd;
unsigned short hVPB;
char far * pBoot;
{
 struct Extended_Boot far *ExtBoot = (struct Extended_Boot far *)pBoot;
15 int i,err = NO_ERROR;

 switch (flag) {
 case MOUNT_NEW:
 /* New volume mounted, check for acceptance */
20
 /* Check signature */

 if (ExtBoot->Boot_Sig < 41)
 err = !NO_ERROR;
25

 /* Check FSD name */

 i=0;
 do {
30 if (ExtBoot->Boot_System_ID[i] != FS_NAME[i]) {
 err = !NO_ERROR;
 break;
 }
 } while (FS_NAME[++i] != '\0');
35

 /* Copy boot record information (if volume is accepted) */

 if (err == NO_ERROR) {
 pvpfsi->vpi_vid = ExtBoot->Boot_Serial;
40 pvpfsi->vpi_bsize = ExtBoot->Boot_BPB.BytesPerSector;
 pvpfsi->vpi_totsec = ExtBoot->Boot_BPB.TotalSectors ?
 (long)ExtBoot->Boot_BPB.TotalSectors :
 ExtBoot->Boot_BPB.Ext_TotalSectors;
 pvpfsi->vpi_trksec = ExtBoot->Boot_BPB.SectorsPerTrack;
45 pvpfsi->vpi_nhead = ExtBoot->Boot_BPB.Heads;
 for (i=0; i<VOLLABELLEN; i++)
 pvpfsi->vpi_text[i] = ExtBoot->Boot_Vol_Label[i];
 pvpfsi->vpi_text[VOLLABELLEN] = '\0';
50 }

 break;

 case MOUNT_REMOVED:
55 /* Volume was removed, but references to it still exists */

```

```

 /* No action needed in this FSD */
 break;

 case MOUNT_DISMOUNT:
5 /* Volume was removed and all references have been closed */

 /* No action needed in this FSD */
 break;

10 case MOUNT_FORMAT:
 /* Accept volume blindly, format to fit this file system */

 /* Format not allowed in this FSD */
 err = !NO_ERROR;
15 break;
 }

 if (Is_Shutdown_Test) Log_SD_Stats(&(pSDdata->Other_stats));
 return(err);
20 }

int far pascal
FS_MOVE(pcdfsi, pcdfsd, pSrc, iSrcCurDirEnd, pDst, iDstCurDirEnd, flags)
25 struct cdfsi far * pcdfsi;
 struct cdfsd far * pcdfsd;
 char far * pSrc;
 unsigned short iSrcCurDirEnd;
 char far * pDst;
 unsigned short iDstCurDirEnd;
30 unsigned short flags;
{
 if (Is_Shutdown_Test) Log_SD_Stats(&(pSDdata->Other_stats));
 return ERROR_NOT_SUPPORTED;
}
35

int far pascal
FS_NEWSIZE(psffsi, psffsd, len, IOflag)
 struct sffsi far * psffsi;
 struct sffsd far * psffsd;
40 unsigned long len;
 unsigned short IOflag;
{
 int rc, index;

45 if (IsEnabled(ENT_NEWSIZE,(int far *)&index)) {
 rc = Disable(ENT_NEWSIZE,index);
 if (rc) return rc;
 psffsi->sfi_size = len;
 return SaveFlags(IOflag);
50 }

 if (Is_Shutdown_Test) Log_SD_Stats(&(pSDdata->Other_stats));
 return ERROR_NOT_SUPPORTED;
}
55

int far pascal

```

```

FS_NMPIPE(psffsi, psffsd, OpType, pOpRec, pData, pName)
struct sffsi far * psffsi;
struct sffsd far * psffsd;
unsigned short OpType;
5 union npper far * pOpRec;
char far * pData;
char far * pName;
{
 int rc, index;
10 unsigned short sel, off, Valid, LSeg=0, LRec = 0, LDat=0, LNam=0;

 if (IsEnabled(ENT_NMPIPE,(int far *)&index)) {
 rc = Disable(ENT_NMPIPE,index);
 if (rc) return rc;
15
 switch (OpType) {
 case NMP_GetPHandState:
 case NMP_SetPHandState:
 LRec = sizeof(struct phs_param);
20 LDat = ((struct phs_param far *)pOpRec)->phs_dlen;
 break;
 case NMP_PipeQInfo:
 LRec = sizeof(struct npi_param);
 LDat = ((struct npi_param far *)pOpRec)->npi_dlen;
25 break;
 case NMP_PeekPipe:
 LRec = sizeof(struct npp_param);
 LDat = ((struct npp_param far *)pOpRec)->npp_dlen;
 break;
30 case NMP_ConnectPipe:
 LRec = sizeof(struct npc_param);
 LDat = ((struct npc_param far *)pOpRec)->npc_dlen;
 break;
 case NMP_DisconnectPipe:
35 case NMP_TransactPipe:
 LRec = sizeof(struct npd_param);
 LDat = ((struct npd_param far *)pOpRec)->npd_dlen;
 break;
 case NMP_READRAW:
 LRec = sizeof(struct npr_param);
40 LDat = ((struct npr_param far *)pOpRec)->npr_dlen;
 break;
 case NMP_WRITERAW:
 LRec = sizeof(struct npw_param);
45 LDat = ((struct npw_param far *)pOpRec)->npw_dlen;
 break;
 case NMP_WAITPIPE:
50 LRec = sizeof(struct npq_param);
 LDat = ((struct npq_param far *)pOpRec)->npq_dlen;
 if (pName != LNULL) {
 /* get the length of the pipe name */
55 while(*(pName+(LNam++)) != '\0');
 }
 }
 }
}

```

```

 break;
 case NMP_CALLPIPE:
 LRec = sizeof(struct npx_param);
 LDat = ((struct npx_param far *)pOpRec)->npx_ilen;
 if (pName != LNULL) {
 /* get the length of the pipe name */
 while(*(pName + (LName++)) != '\0');
 }
 break;
 case NMP_QNmPipeSemState:
 LRec = sizeof(struct qnps_param);
 LDat = ((struct qnps_param far *)pOpRec)->qnps_dlen;
 break;

 default:
 return FSNMP_UNKNOWN_OPTYPE;
}

LSeg = (sizeof(short) * 7) + sizeof(struct sffsi)
 + sizeof(struct sffsd)
 + sizeof(union npoper) + LDat + LName;

rc = InitTCBInstanceData((unsigned short far *)&sel,
 (unsigned short far *)&off,
 (int far *)&index, LSeg);
if (rc || (index == -1)) goto abort;

/* copy sffsi struct if valid */
Valid = (psffsi == LNULL) ? INVALID : VALID;
rc = CopyParam(sel, (unsigned short far *)&off,
 sizeof(short), (char far *)&Valid);
if (rc) goto abort;

if (Valid) {
 rc = CopyParam(sel, (unsigned short far *)&off,
 sizeof(struct sffsi), (char far *)&psffsi);
 if (rc) goto abort;
}
else { /* always advance the pointer */
 off += sizeof(struct sffsi);
}

/* copy sffsd struct if valid */
Valid = (psffsd == LNULL) ? INVALID : VALID;
rc = CopyParam(sel, (unsigned short far *)&off,
 sizeof(short), (char far *)&Valid);
if (rc) goto abort;

if (Valid) {
 rc = CopyParam(sel, (unsigned short far *)&off,
 sizeof(struct sffsd), (char far *)&psffsd);
 if (rc) goto abort;
}
else { /* always advance the pointer */
 off += sizeof(struct sffsd);
}

```

```

/* copy OpType */
5 rc = CopyParam(sel,(unsigned short far *)&off,
 sizeof(short),(char far *)&OpType);
 if (rc) goto abort;

/* copy the data record which varies for each OpType */
10 Valid = (pOpRec == LNULL) ? INVALID : VALID;
 rc = CopyParam(sel,(unsigned short far *)&off,
 sizeof(short),(char far *)&Valid);
 if (rc) goto abort;

 if (Valid) {
15 rc = CopyParam(sel,(unsigned short far *)&off,
 LRec,(char far *)pOpRec);
 off += sizeof(union npoper) - LRec;
 if (rc) goto abort;
 }
 else { /* always advance the pointer */
20 off += sizeof(union npoper);
 }

/* copy data buffer if valid */
25 Valid = (pData == LNULL) ? INVALID : VALID;
 rc = CopyParam(sel,(unsigned short far *)&off,
 sizeof(short),(char far *)&Valid);
 if (rc) goto abort;

 if (Valid) {
30 rc = CopyParam(sel,(unsigned short far *)&off,
 LDat,(char far *)pData);
 if (rc) goto abort;
 }
 else { /* always advance the pointer */
35 off += LDat;
 }

/* copy named pipe name */
40 Valid = (pName == LNULL) ? INVALID : VALID;
 rc = CopyParam(sel,(unsigned short far *)&off,
 sizeof(short),(char far *)&Valid);
 if (rc) goto abort;

 if (Valid) {
45 rc = CopyParam(sel,(unsigned short far *)&off,
 LNam,(char far *)pName);
 if (rc) goto abort;
 }
50

/* no errors occurred */
 return NO_ERROR;
}

55 /* this entry point not enabled to save data */
 if (Is_Shutdown_Test) Log_SD_Stats(&(pSDdata->Other_stats));

```

```

 return ERROR_NOT_SUPPORTED;

 abort:

 DeleteTabEntry();
 return rc;

 }

 int far pascal
 FS_FINDNOTIFYCLOSE(handle)
 unsigned short handle;
 {
 if (Is_Shutdown_Test) Log_SD_Stats(&(pSDdata->Other_stats));
 return ERROR_NOT_SUPPORTED;
 }

 int far pascal
 FS_FINDNOTIFYFIRST(pcdfsi, pcdfsd, pName, iCurDirEnd, attr, handle, pData,
 cbData, pcMatch, level, timeout)
 struct cdfsi far * pcdfsi;
 struct cdfsd far * pcdfsd;
 char far * pName;
 unsigned short iCurDirEnd;
 unsigned short attr;
 unsigned short far * handle;
 char far * pData;
 unsigned short cbData;
 unsigned short far * pcMatch;
 unsigned short level;
 unsigned long timeout;
 {
 if (Is_Shutdown_Test) Log_SD_Stats(&(pSDdata->Other_stats));
 return ERROR_NOT_SUPPORTED;
 }

 int far pascal
 FS_FINDNOTIFYNEXT(handle, pData, cbData, pcMatch, infolevel, timeout)
 unsigned short handle;
 char far * pData;
 unsigned short cbData;
 unsigned short far * pcMatch;
 unsigned short infolevel;
 unsigned long timeout;
 {
 if (Is_Shutdown_Test) Log_SD_Stats(&(pSDdata->Other_stats));
 return ERROR_NOT_SUPPORTED;
 }

 int far pascal
 FS_PROCESSNAME(pNameBuf)
 char far *pNameBuf;
 {
 if (Is_Shutdown_Test) Log_SD_Stats(&(pSDdata->Other_stats));
 return NO_ERROR;
 }

```

```

int far pascal
FS_RMDIR(pcdfsi, pcdfsd, pName, iCurDirEnd)
 struct cdfsi far * pcdfsi;
 struct cdfsd far * pcdfsd;
5 char far * pName;
 unsigned short iCurDirEnd;
 {
 if (Is_Shutdown_Test) Log_SD_Stats(&(pSDdata->Other_stats));
 return NO_ERROR;
10 }

int far pascal
FS_SETSWAP(psffsi, psffsd)
 struct sffsi far * psffsi;
15 struct sffsd far * psffsd;
 {
 if (Is_Shutdown_Test) Log_SD_Stats(&(pSDdata->Other_stats));
 return ERROR_NOT_SUPPORTED;
 }
20

/*=====
=====*/
/**** Getvpfsd - copy vpfsd data to TCB instance data area
*
25 * ENTRY hVPB - volume parameter block handle
* EXIT -none-
* RETURN NO_ERROR if successful
*
* this routine obtains a TCB instance data buffer, then copies the
30 * vpfsd data it gets from FSH_GETVOLPARM into it.
*
* WARNING:
* EFFECTS:
*/
35 int Getvpfsd(hVPB)
 unsigned short hVPB;
 {
 int rc,index;
 unsigned short sel, off, LSeg;
40 struct vpfsi far * pVPBfsi;
 struct vpfsd far * pVPBfsd;

 LSeg = sizeof(vpfsdRec);
 rc = InitTCBInstanceData((unsigned short far *)&sel,
45 (unsigned short far *)&off,
 (int far *)&index,LSeg);
 if (rc) goto abort;

 rc = FSH_GETVOLPARM(hVPB,(struct vpfsi far * far *)&pVPBfsi,
50 (struct vpfsd far * far *)&pVPBfsd);
 if (rc) goto abort;

 rc = CopyParam(sel,(unsigned short far *)&off,sizeof(struct vpfsd),
 (char far *) (pVPBfsd->vpd_work));
55 if (rc) goto abort;

```



```

 return NO_ERROR;
abort:
 DeleteTabEntry();
 return rc;
5 }
 /*=====
=====*/
 /*** SaveFlags -- save the IOflag parameter into this thread's data buffer
 *
10 * ENTRY flags - IOflag parameter from various FS_ entries
 * EXIT -none-
 * RETURN NO_ERROR if successful
 *
 * this routine obtains a TCB instance data buffer, then copies the
15 * flags into it.
 *
 * WARNING:
 * EFFECTS:
 */
20 int SaveFlags(flags)
 unsigned short flags;
 {
 int rc,index;
 unsigned short sel, off, LSeg;
25
 LSeg = sizeof(BitRec);
 rc = InitTCBInstanceData((unsigned short far *)&sel,
 (unsigned short far *)&off,
 (int far *)&index,LSeg);
30
 if (rc) goto abort;

 rc = CopyParam(sel,(unsigned short far *)&off,sizeof(short),
 (char far *)&flags);
 if (rc) goto abort;
35
 return NO_ERROR;
abort:
 DeleteTabEntry();
 return rc;
40 }

```

## APPENDIX V

```

LIBRARY
DESCRIPTION 'BASEFSD - R3ENTRYNEW testing'

5 ; NOTE: The FSD loader only supports the following rules:
 ; CODE PRELOAD
 ; DATA PRELOAD [SINGLE | NONE] SHARED MOVABLE
 NONDISCARDABLE
 ;
10 ;

 CODE PRELOAD
 DATA PRELOAD SINGLE SHARED MOVABLE
 PROTMODE

15

 ;IMPORTS
 ; FSHELPER.

20 EXPORTS

 FS_ATTRIBUTE= FS_ATTRIBUTE ; DWORD attribute vector
 FS_NAME= _FS_NAME ; ASCIIZ name string

 FS_ATTACH ; From FS_ATTACH on are procedures
25 FS_CHDIR
 FS_CHGFILEPTR
 FS_CLOSE
 FS_COMMIT
 FS_COPY
30 FS_DELETE
 FS_EXIT
 FS_FILEATTRIBUTE
 FS_FILEINFO
 FS_FILEIO
35 FS_FINDCLOSE
 FS_FINDFIRST
 FS_FINDFROMNAME
 FS_FINDNEXT
 FS_FINDNOTIFYCLOSE
40 FS_FINDNOTIFYFIRST
 FS_FINDNOTIFYNEXT
 FS_FLUSHBUF
 FS_FSCTL
 FS_FSINFO
45 FS_INIT
 FS_IOCTL
 FS_MKDIR
 FS_MOUNT
 FS_MOVE
50 FS_NEWSIZE
 FS_NMPIPE
 FS_OPENCREATE
 FS_PATHINFO
 FS_PROCESSNAME
55 FS_READ
 FS_RMDIR

```

|    |                |
|----|----------------|
|    | FS_SETSWAP     |
|    | FS_SHUTDOWN    |
|    | FS_WRITE       |
| 5  | MFS_CHGFILEPTR |
|    | MFS_CLOSE      |
|    | MFS_INIT       |
|    | MFS_OPEN       |
|    | MFS_READ       |
| 10 | MFS_TERM       |

## APPENDIX VI

```

/* BASEFSD.H
 * Sue Adams
 * Copyright 1988, Microsoft Corp.
5 *
 * MODIFICATION HISTORY:
 * 89.04.07 -- integrate various .h files
 * 88.10. -- initial version
 */
10 /*
 * names for the different FSD's
 */

15 #if defined(generic)

#define FSDNAME "GENREM"
#define VERSION "Ver 2.0 (89.04.06)\r\n";
#define SIGNONMSG "Multi-Threaded REMOTE/FILIO FSD\r\n";
20 unsigned long FS_ATTRIBUTE = FSA_REMOTE|FSA_LOCK;

#elif defined(fsnmpipe)

#define FSDNAME "UNC1"
25 #define VERSION "Ver 2.0 (89.04.05)\r\n";
#define SIGNONMSG "UNC remote pipe test FSD\r\n";
unsigned long FS_ATTRIBUTE = FSA_REMOTE|FSA_UNC;

#elif defined(onetid)
30 #define FSDNAME "REM1"
#define VERSION "Ver 2.0 (89.04.06)\r\n"
#define SIGNONMSG "Single Threaded REMOTE FSD\r\n"
unsigned long FS_ATTRIBUTE = FSA_REMOTE;
35 #endif

/*
 * FSH_PROBEBUF operation codes
40 */

#define READPROBE 0
#define WRITEPROBE 1

45

/*
 * defines for FS_CLOSE
 */

50 #define NOT_FINAL_CLOSE 0x0000
#define FINAL_CLOSE_THIS_PROC 0x0001
#define FINAL_CLOSE_ALL_PROC 0x0002

/*
55 * defines for sfi_type
 */

```

```

/*
#define STYPE_FILE 0x0000
#define STYPE_DEVICE 0x0001
#define STYPE_NMPIPE 0x0002
5 #define STYPE_FCB 0x0004
*/

/*
* defines for FSH_QSYSINFO
10 */

#define GET_MAX_SECTOR_SIZE 1
#define GET_PID 2
#define GET_TID 3
15

/*
* Definitions for OPEN action codes
*/

20 #define FILE_EXISTS 0x0001
#define FILE_CREATED 0x0002
#define FILE_REPLACED 0x0003
/* FSD will know to return ERROR_OPEN_FAILED if OpenAction is the following */
#define UNDEFINED_ACTION 0xffff
25 /*
* Defines for fGenNeedEA parameter in FS_OPENCREATE
*/
#define NEED_EAS 0x0001 /* fGenNeedEA */
#define NICE_EAS 0x0000
30

/*
*
* Global Definitions
*/

35 #undef NULL
#define NULL 0
#define LNULL 0L

#define INVALID 0
40 #define VALID 1

#define SEM_TIMEOUT 5000L

#define BOGUS_CTIME 0x1111
45 #define BOGUS_CDATE 0x2222
#define BOGUS_ETIME 0x3333
#define BOGUS_ADATE 0x4444
#define BOGUS_MTIME 0x5555
#define BOGUS_MDATE 0x6666
50 #define BOGUS_SIZE 0x0001
#define BOGUS_POS 0x0000
#define BOGUS_WORK_CHAR 'Q'

#define DEVFSD_MSG "abc"
55 #define VPFSD_MSG "here is some vpfsd data"

```

```

/*
 * Useful Macros
 */

5 #define MAKEFP(s,o) ((void far *)((unsigned long)((o) | ((unsigned long)(s)) < <16)))

 #define FIELDOFFSET(type,field) ((unsigned int)&(((type *)0)->field))

 #define ISBETWEEN(l,h,x) ((l <= x) && (x <= h))

10 /*
 * kernel defines for FS_FSCTL
 */

15 #define GET_ERR_INFO 1 /* func value */
 #define GET_MIN_MAX_EA 2 /* func value */

 #define ROUTE_BY_NAME 3 /* iArgType value */
 #define BAD_HANDLE -1

20 /*
 * enumerate the entrypoints which will save data
 */
 enum entrypoints {

25 ENT_FILEINFO,
 ENT_PATHINFO,
 ENT_FSCTL,
 ENT_NMPIPE,
 ENT_OPENCREATE,
30 ENT_FSINFO,
 ENT_ATTACH,
 ENT_FLUSHBUF,
 ENT_WRITE,
 ENT_READ,
35 ENT_NEWSIZE,
 ENT_FILEIO,
 ENT_COMMIT,
 ENT_CLOSE,
 ENT_CHGFILEPTR,
40 ENT_COUNT /* this must be last in the list! */
 };

/*
 * FSD Specific Error Codes (0xef00 - 0xefff) (always treat as unsigned)
 * =====
45 */
/* when adding a new err message, update the table of strings in fsdtools.c */
#define START_FSD_ERROR_CODES 0xef00

50 #define FAILURE START_FSD_ERROR_CODES + 0
 #define FSCTL_UNKNOWN_FUNCTION START_FSD_ERROR_CODES + 1
 #define FSCTL_NO_SELECTOR START_FSD_ERROR_CODES + 2
 #define FSCTL_NO_TAB_ENTRY START_FSD_ERROR_CODES + 3
 #define FSCTL_BUF_OVERFLOW START_FSD_ERROR_CODES + 4
55 #define FSNMP_UNKNOWN_OPTYPE START_FSD_ERROR_CODES + 5
 #define TCB_DATA_EXISTS START_FSD_ERROR_CODES + 6

```

```

* The following structure is used by fsd232.c
*/
typedef struct { /* recovery buffer is cast to this type */
 unsigned length;
5 unsigned IOflag;
} BitRec;

typedef struct {
 unsigned short length;
10 unsigned short owner;
 unsigned short flag;
 struct sffsi thesffsi;
 struct sffsd thesffsd;
 unsigned short level;
15 char far * pData; /* only verify the pointer */
 unsigned short cbData;
 unsigned short IOflag;
} FileInfoRec;

20 typedef struct {
 unsigned short length;
 unsigned short owner;
 unsigned short flag;
 struct cdfsi thecdfsi;
25 struct cdfsd thecdfsd;
 char pName[MAXPATHLEN];
 unsigned short iCurDirEnd;
 unsigned short level;
 char far * pData; /* only verify the pointer */
30 unsigned cbData;
} PathInfoRec;

typedef struct {
 unsigned short length;
35 unsigned short owner;
 struct sffsi thesffsi;
 struct sffsd thesffsd;
 struct cdfsi thecdfsi;
 struct cdfsd thecdfsd;
40 char pName[MAXPATHLEN];
 unsigned short iCurDirEnd;
 unsigned short iArgType;
 unsigned short func;
 char far * pParm; /* only verify the pointer */
45 unsigned short lenParm;
 unsigned short far * plenParmOut;
 char far * pData; /* only verify the pointer */
 unsigned short lenData;
 unsigned short far * plenDataOut;
50 } FsctlRec;

typedef struct {
 unsigned short length;
55 unsigned short owner;
 unsigned short type;
 unsigned short IOflag;

```

```

}CommitnCloseRec;

/* FS_SHUTDOWN test */

5 #define BEFORE_SD 1
 #define DURING_SD 2
 #define AFTER_SD 3
 /* type param values to FS_SHUTDOWN */
 #define SHUTDOWN_START 0
10 #define SHUTDOWN_END 1

 struct SD_Gen_stats { /* these are first in the GDT block */
 unsigned short SD_tid;
 unsigned short FS_SD_bad_type;
15 unsigned long FS_SD_bad_reserved;
 unsigned long Signal_SD_RAM_sem;
 };

 struct SD_FS_stats {
20 unsigned SD_tid_cnt;
 unsigned Other_tid_cnt;
 unsigned Attempt_cnt;
 unsigned Before_cnt;
 unsigned During_cnt;
25 unsigned After_cnt;
 };

 struct SD_stats {
30 struct SD_Gen_stats General_stats;
 struct SD_FS_stats Shutdown_stats;
 struct SD_FS_stats Flushbuf_stats;
 struct SD_FS_stats Commit_stats;
 struct SD_FS_stats Open_stats;
 struct SD_FS_stats Read_stats;
35 struct SD_FS_stats Write_stats;
 struct SD_FS_stats Delete_stats;
 struct SD_FS_stats Other_stats;
40 };

 We claim:

```



```

#define TCB_TABLE_FULL START_FSD_ERROR_CODES + 7
#define FS_ENTRY_ENABLED START_FSD_ERROR_CODES + 8
#define FS_ENTRY_TAB_FULL START_FSD_ERROR_CODES + 9
#define FSD_BUFFER_OVERFLOW START_FSD_ERROR_CODES + 0xa
5 #define FSD_FEALIST_TOO_LONG START_FSD_ERROR_CODES + 0xb
#define FEA_SEGMENT_EXISTS START_FSD_ERROR_CODES + 0xc
#define NULL_FEA_SELECTOR START_FSD_ERROR_CODES + 0xd
/* this value must ALWAYS be defined as the last error code above */
#define END_FSD_ERROR_CODES NULL_FEA_SELECTOR
10

/*
 * FSD Specific Function Codes for FS_FSCtl (0x8000 - ?) (cast as unsigned)
 * Function codes 0x0000-0x7fff are reserved
 */
15 =====
 ==
 */
 /* when adding a new function, update the table of strings in fsdtools.c */
#define START_FSD_FUNC_CODES 0x8000
20

#define FUNC_CLEAR_TABLE START_FSD_FUNC_CODES + 0
#define FUNC_GET_BUF START_FSD_FUNC_CODES + 1
#define FUNC_DEL_BUF START_FSD_FUNC_CODES + 2
#define FUNC_SEL_TAB_DUMP START_FSD_FUNC_CODES + 3
25 #define FUNC_TCB_BUF_COUNT START_FSD_FUNC_CODES + 4
#define FUNC_ENABLE START_FSD_FUNC_CODES + 5
#define FUNC_INIT_FEA_SEG START_FSD_FUNC_CODES + 6
#define FUNC_DEL_FEALIST START_FSD_FUNC_CODES + 7
#define FUNC_SET_SHUTDOWN_DATA START_FSD_FUNC_CODES + 8
30 #define FUNC_SIGNAL_SHUTDOWN START_FSD_FUNC_CODES + 9
#define FUNC_DISABLE START_FSD_FUNC_CODES + 0xa
#define FUNC_SET_EATYPE START_FSD_FUNC_CODES + 0xb
#define FUNC_SET_OPENACTION START_FSD_FUNC_CODES + 0xc
/* this value must ALWAYS be equal to the last defined function above */
35 #define END_FSD_FUNC_CODES FUNC_SET_OPENACTION

/*
 * The following 2 structures are used by fsd231.c
 * data buffer is cast as these types
40 *
 * xxvalid verifies that a certain parameter is not null
 * FS_ATTACH: verifies 3rd parameter
 * FS_OPENCREATE: verifies 1st parameter
 */
45 typedef struct { /* attaching a device or drive */
 unsigned length;
 unsigned xxvalid;
 char ddevfsd[sizeof(long)]; /* used by opencreate */
 } GenericRec;
50

 typedef struct { /* checking HVPB */
 unsigned length;
 struct vpfsd dvpfds;
 } vpfsdRec;
55

/*

```

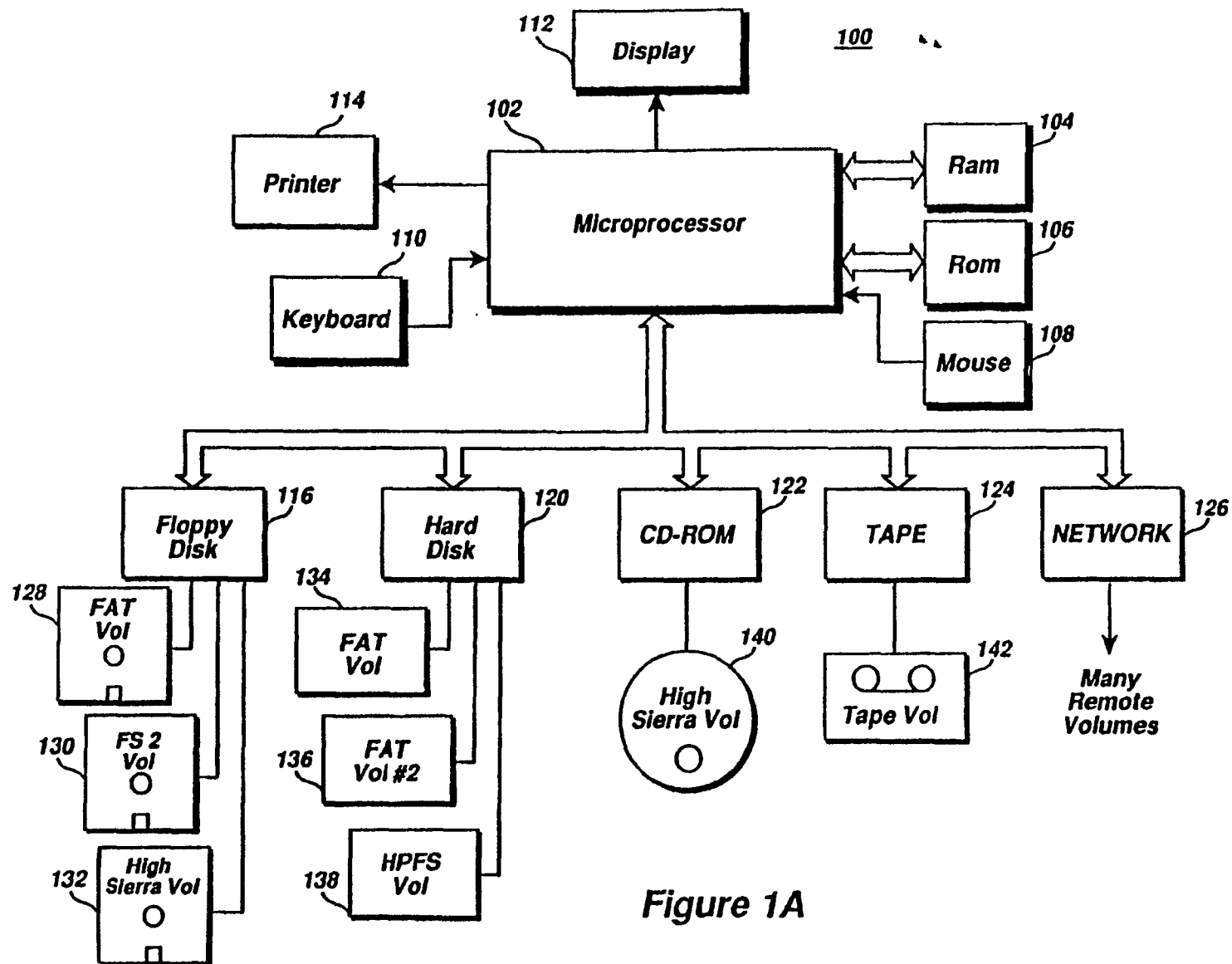


Figure 1A

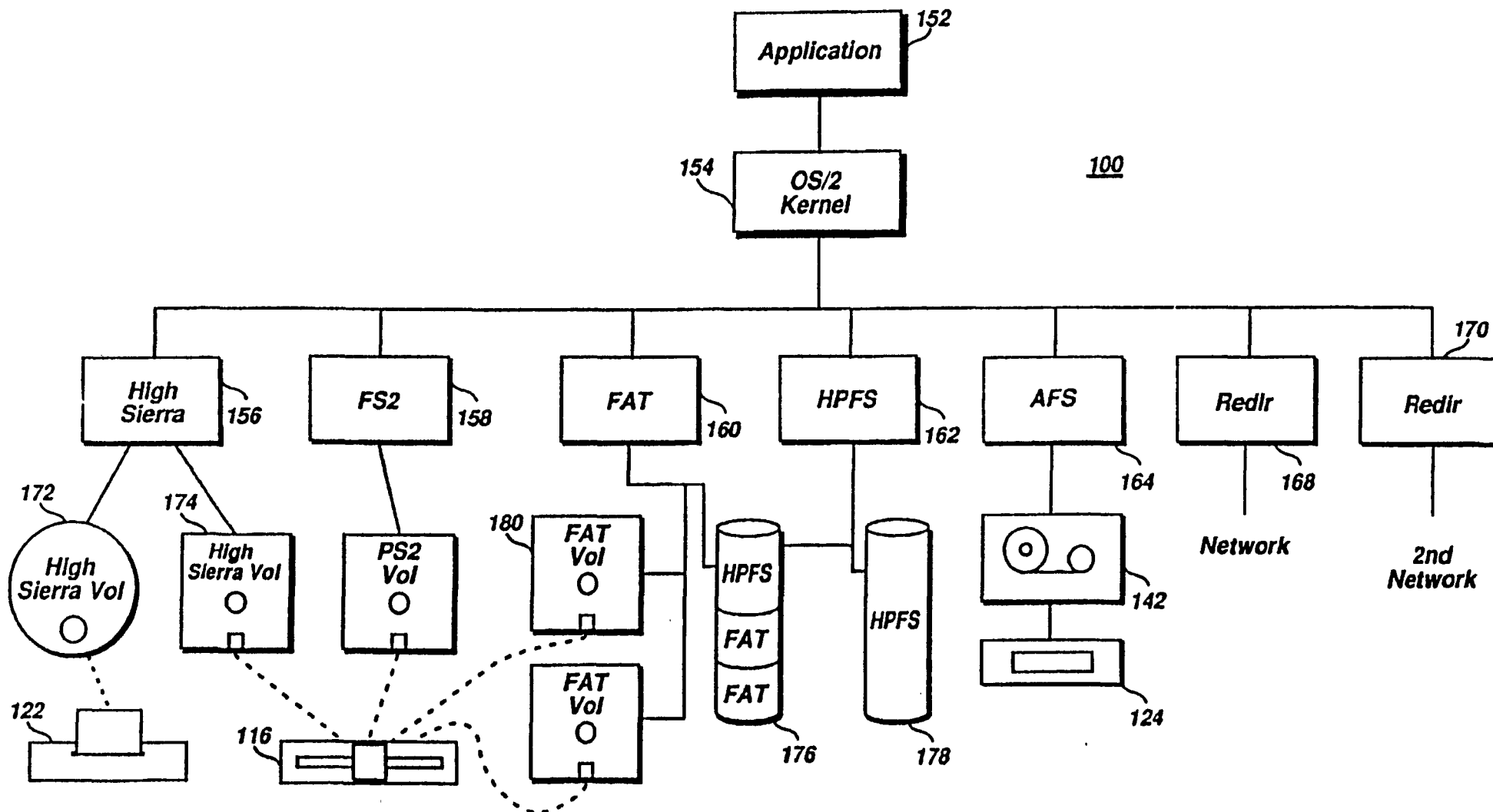


Figure 1B

EP 0 415 346 A2

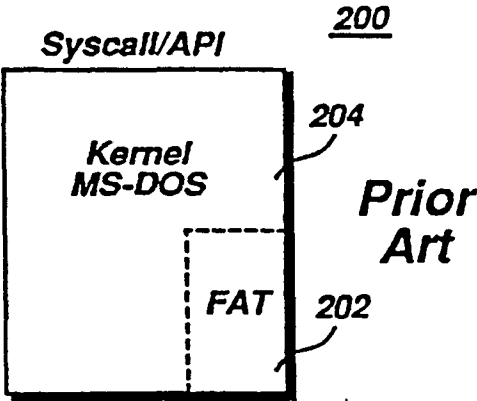


Figure 2A

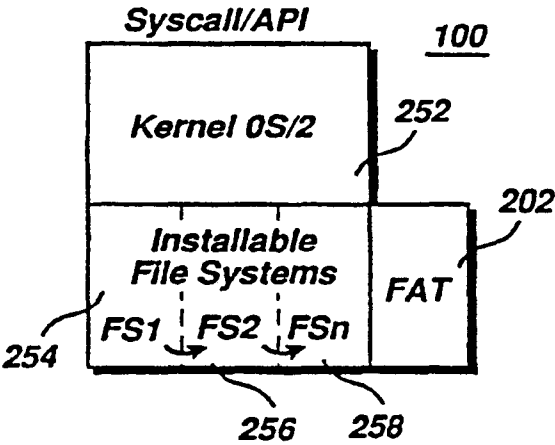


Figure 2B

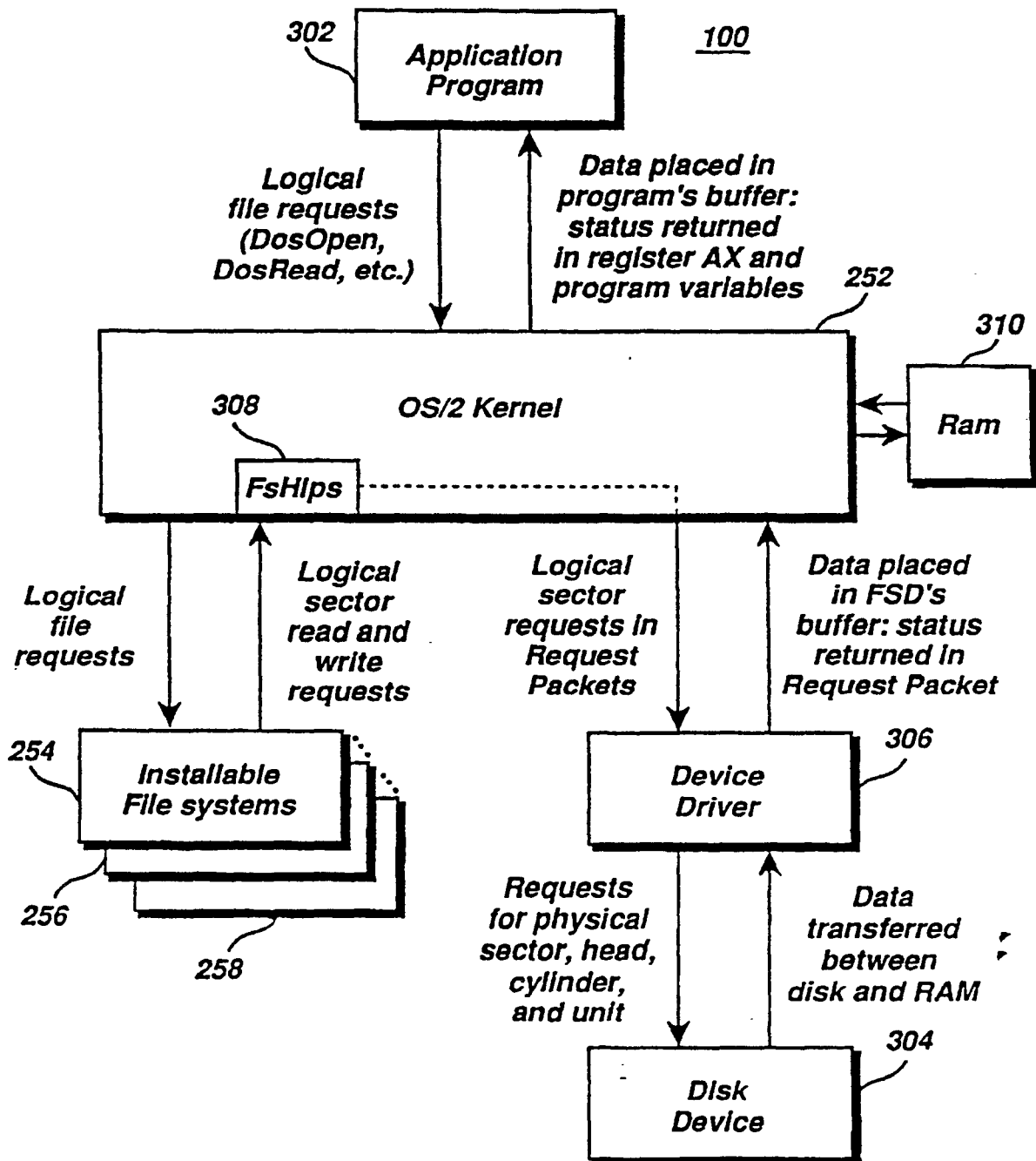
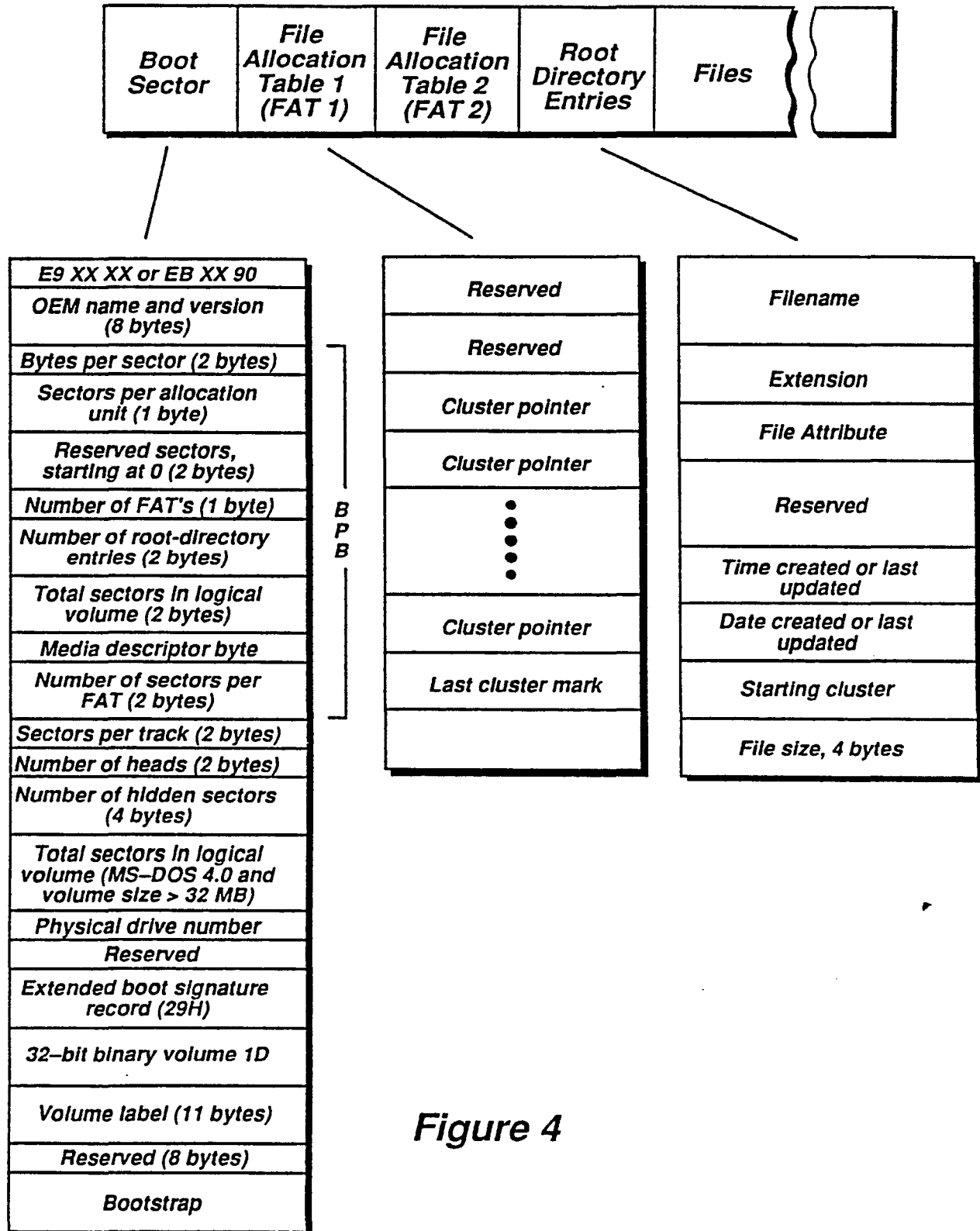


Figure 3

**FAT File System****Figure 4**

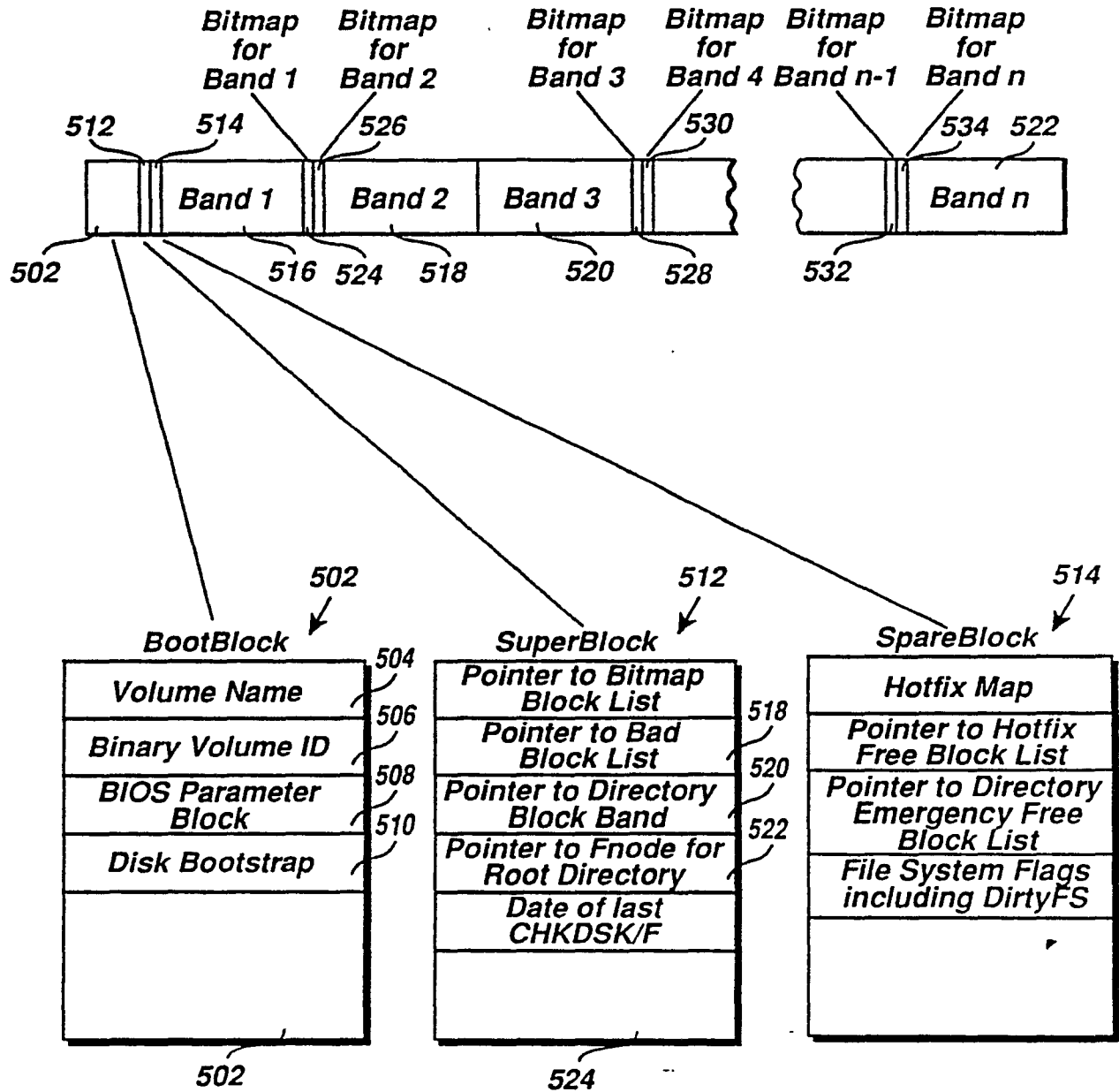
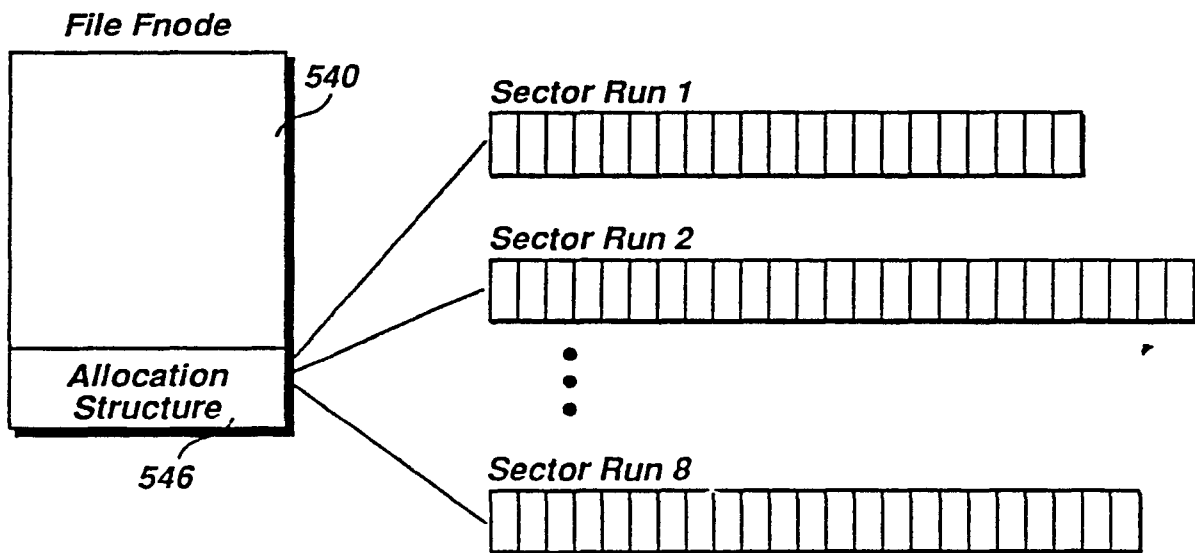
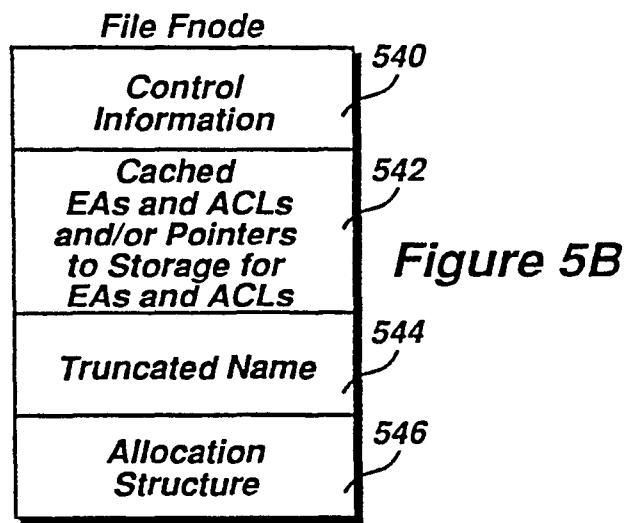


Figure 5A



*Figure 5C*



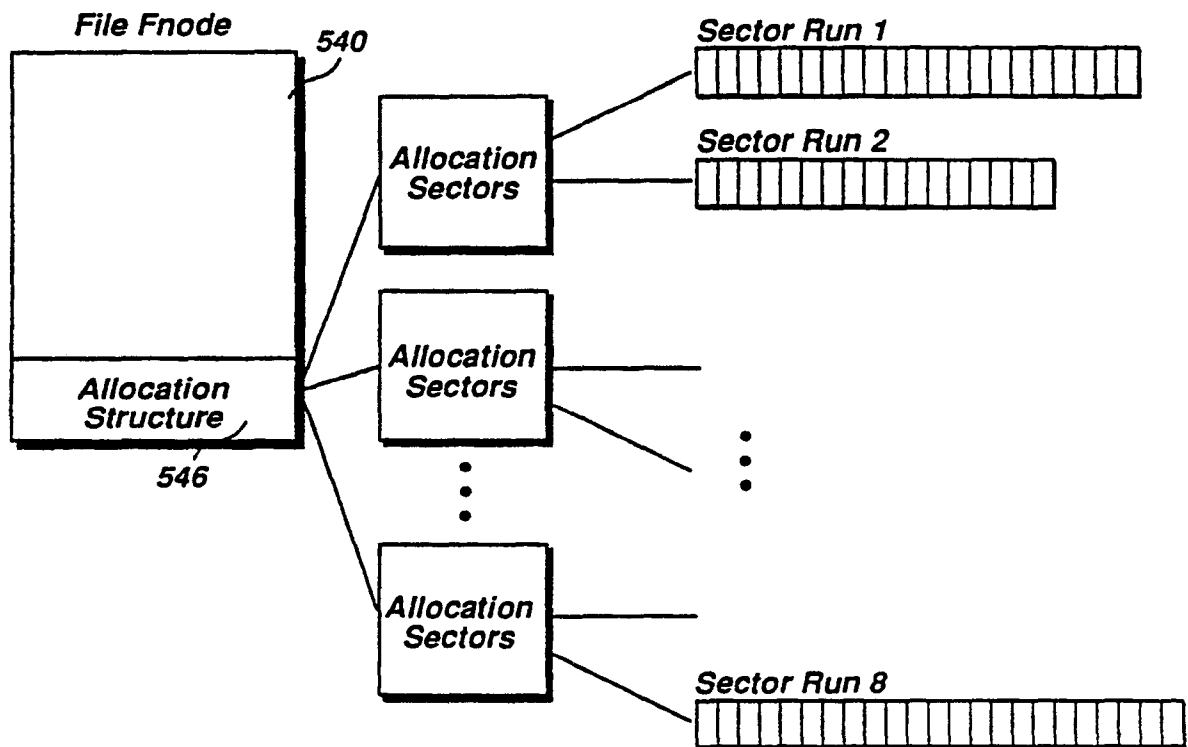


Figure 5D

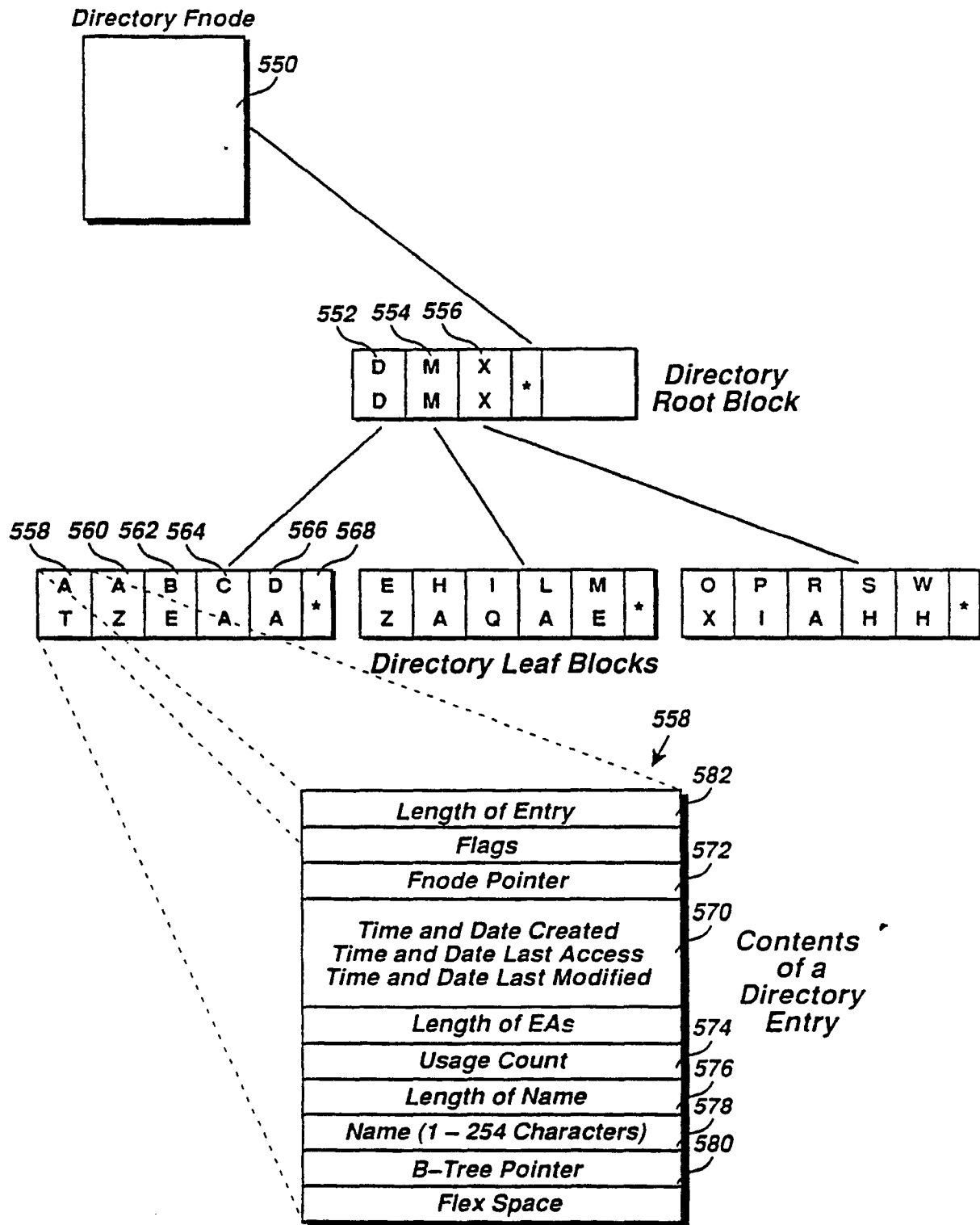
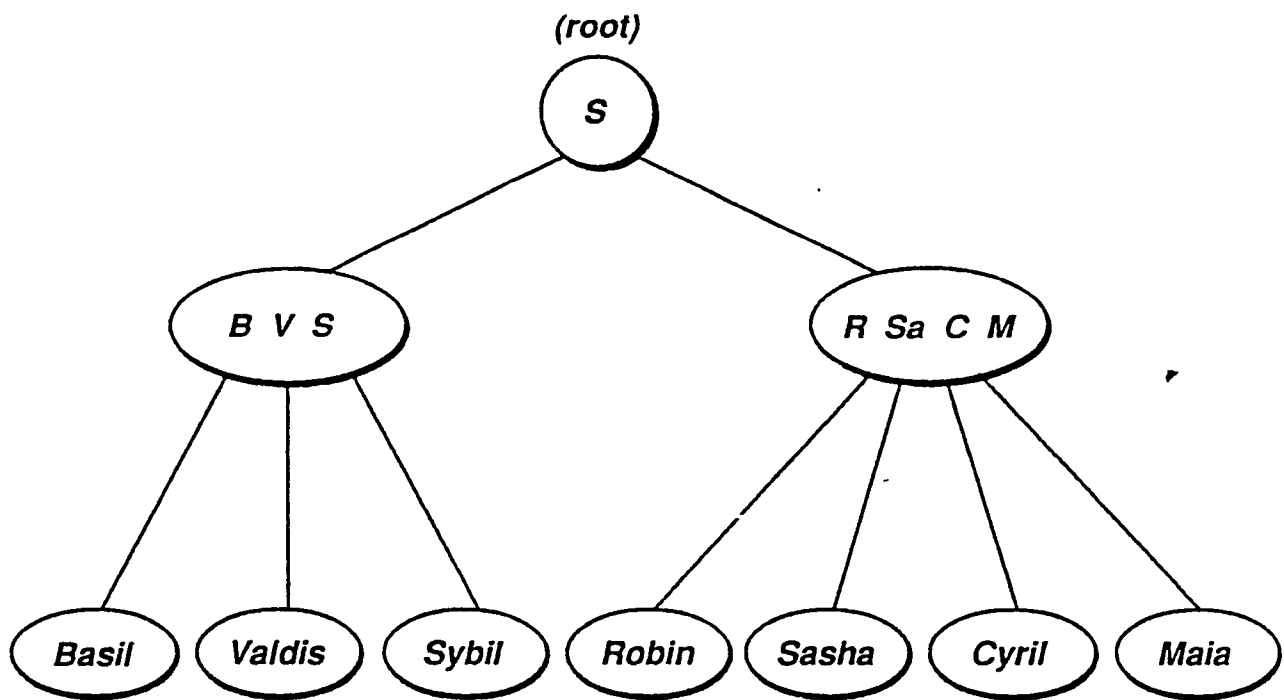
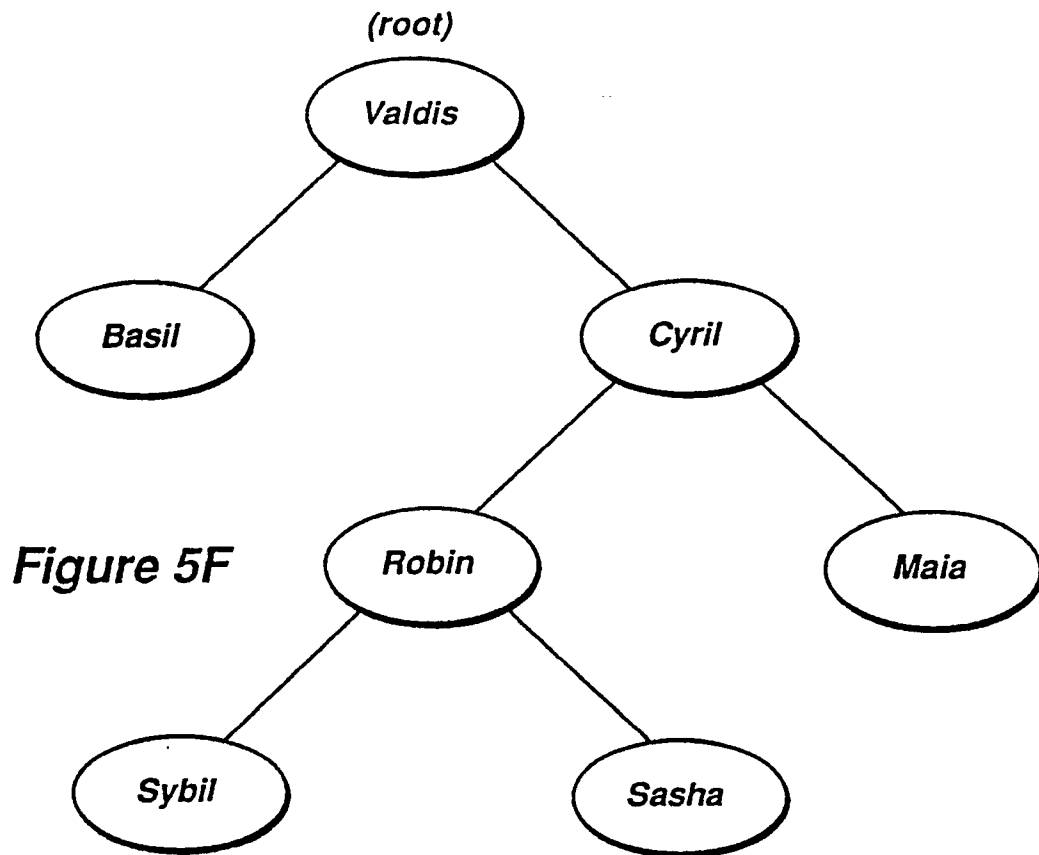
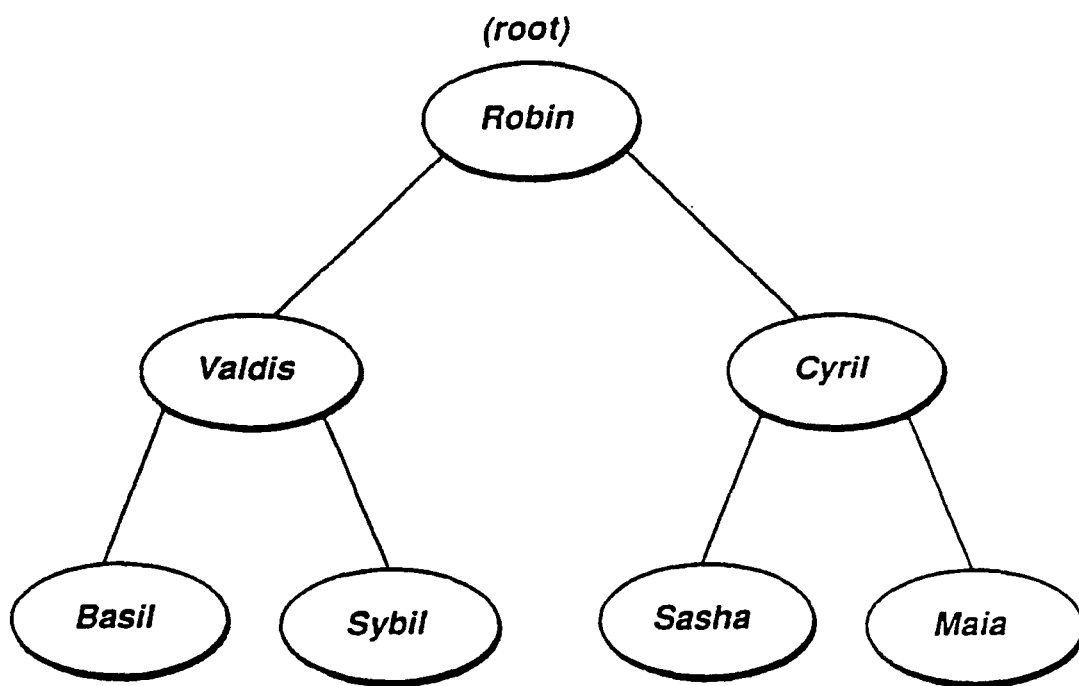


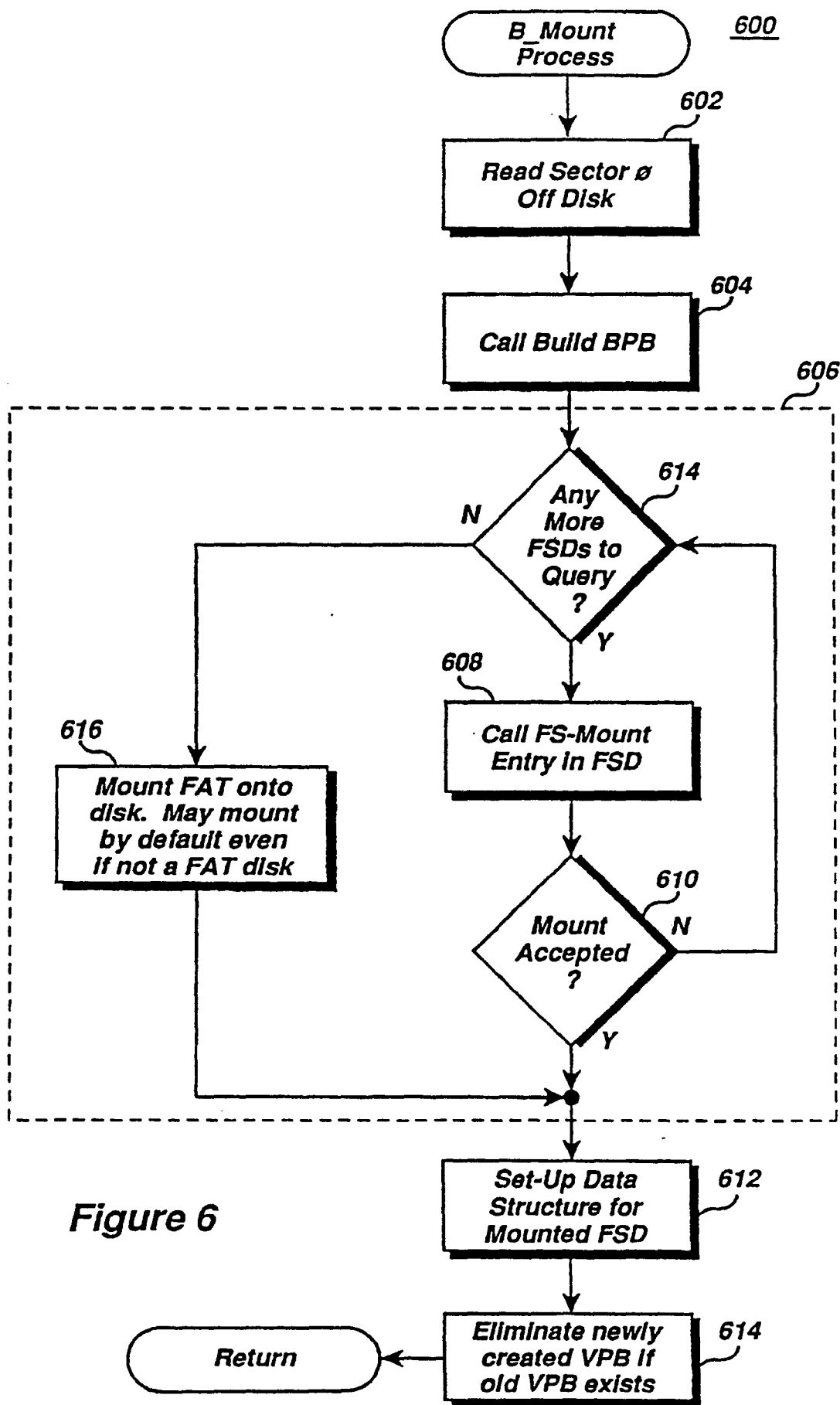
Figure 5E

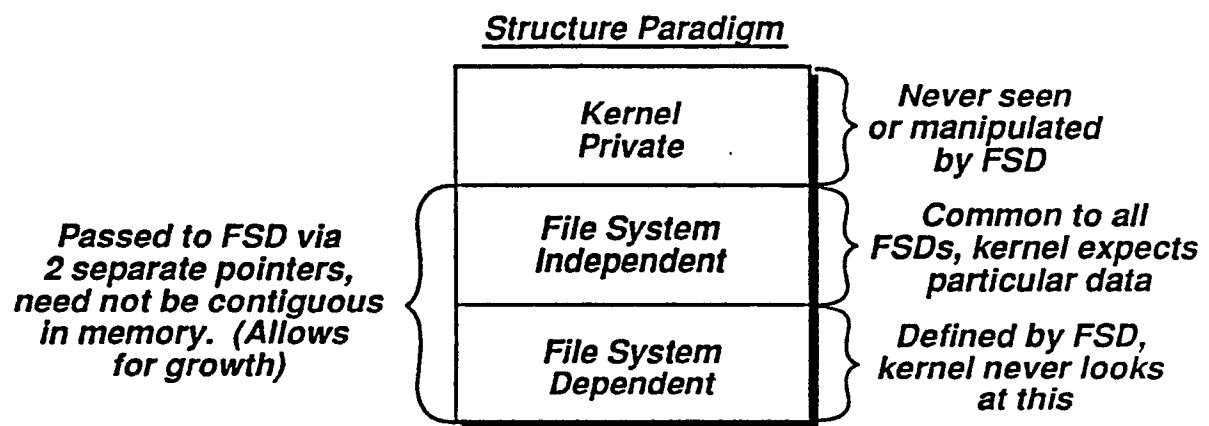




**Figure 5H**

EP 0 415 346 A2





**Figure 7**

EP 0 415 346 A2

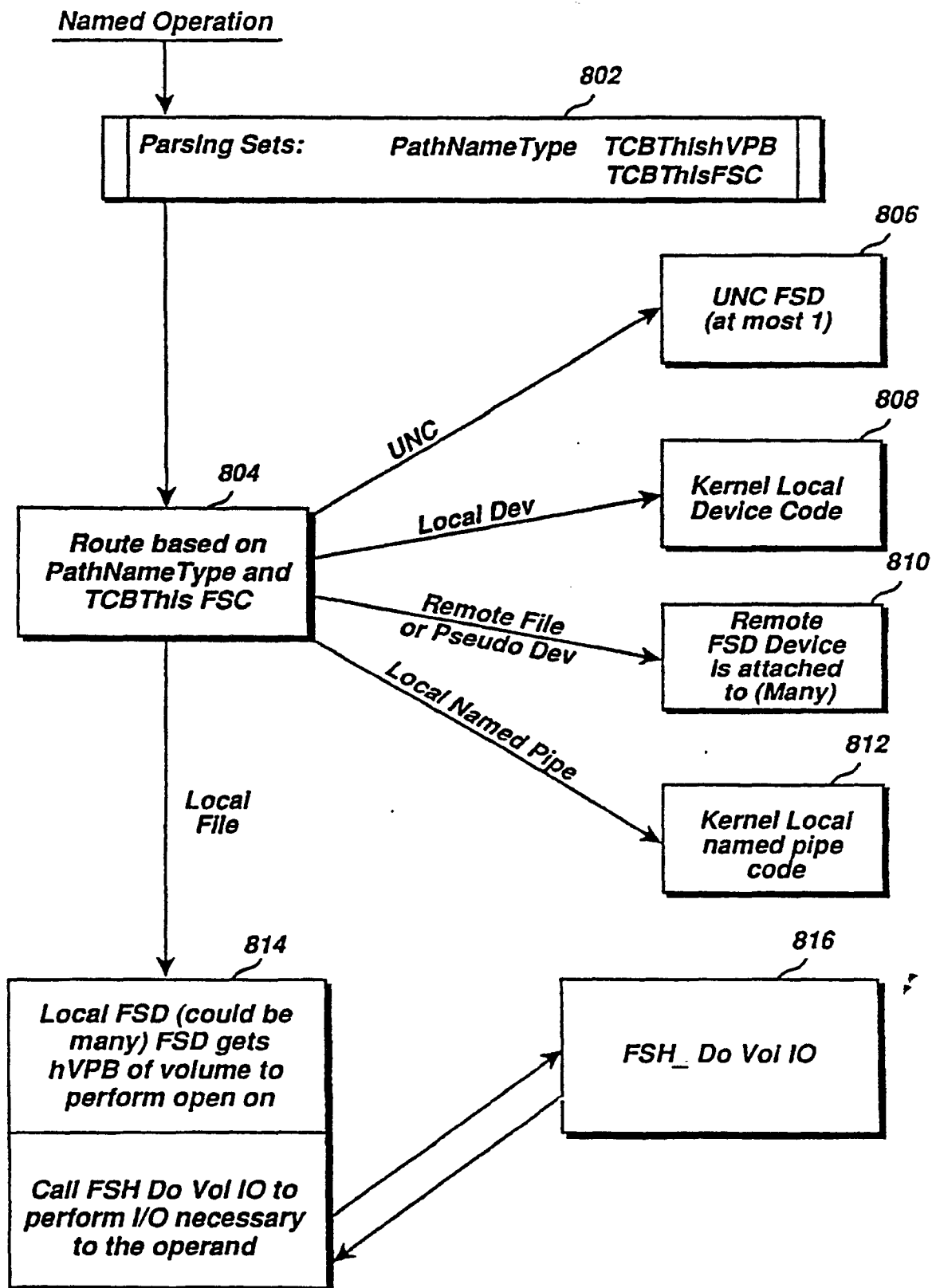


Figure 8

EP 0 415 346 A2

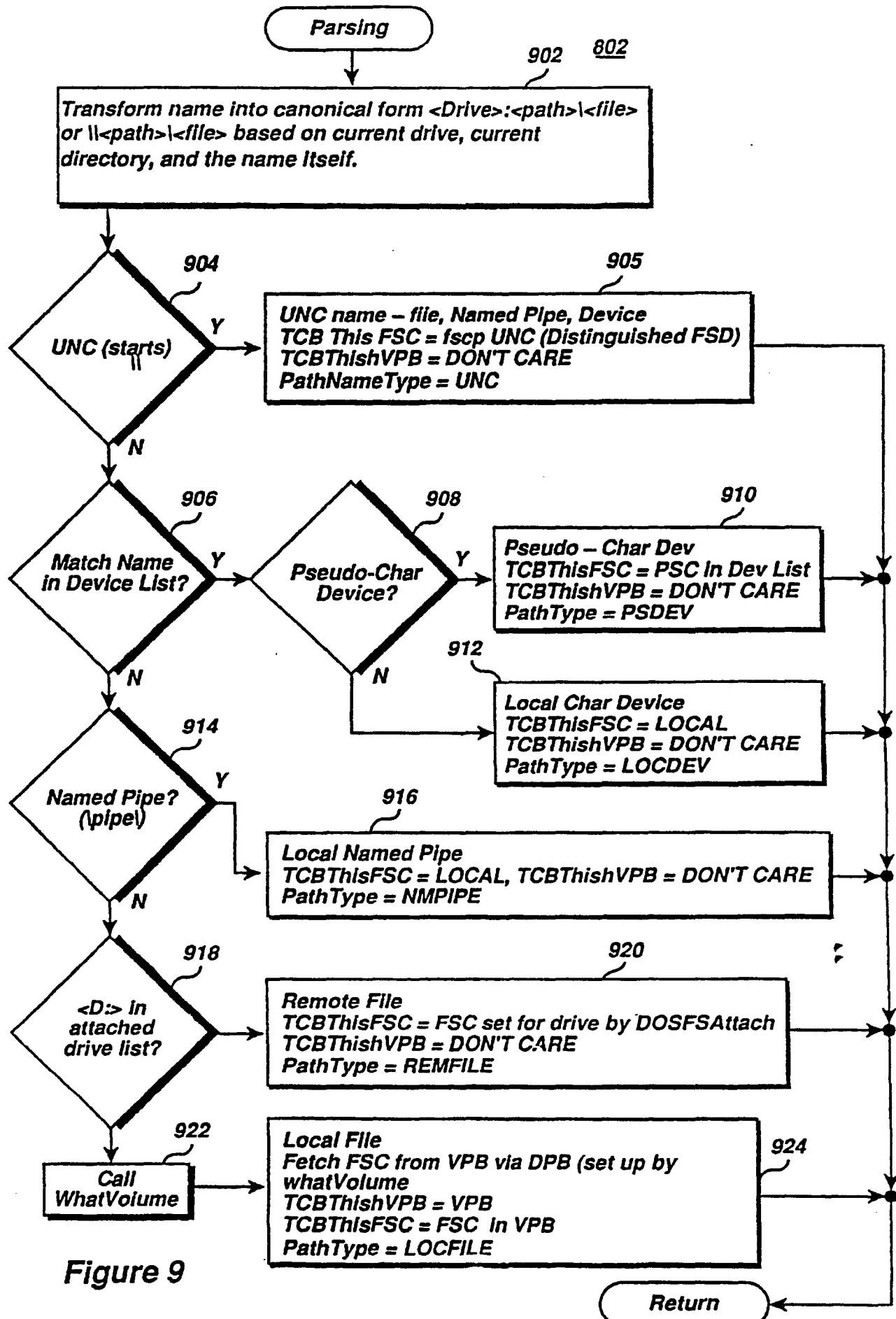


Figure 9



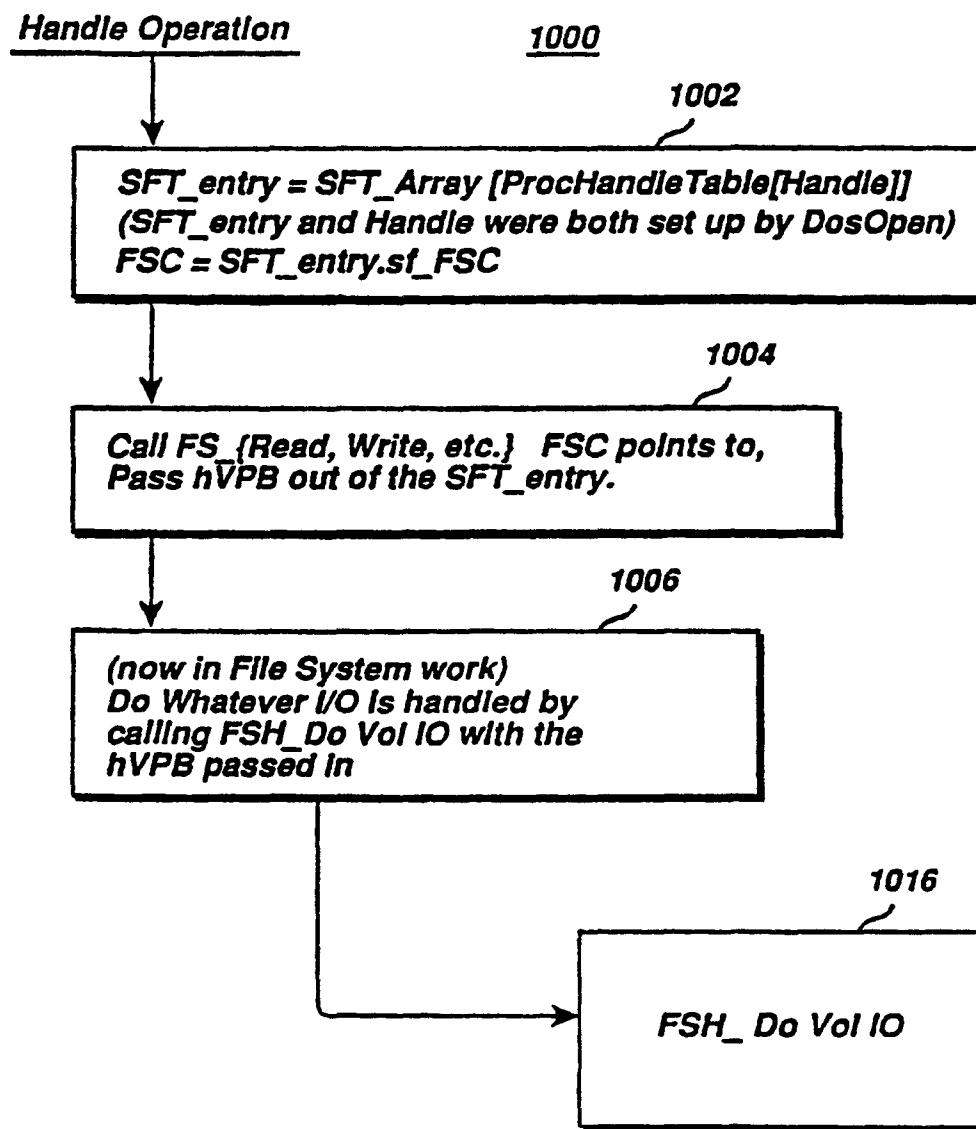


Figure 10

EP 0 415 346 A2

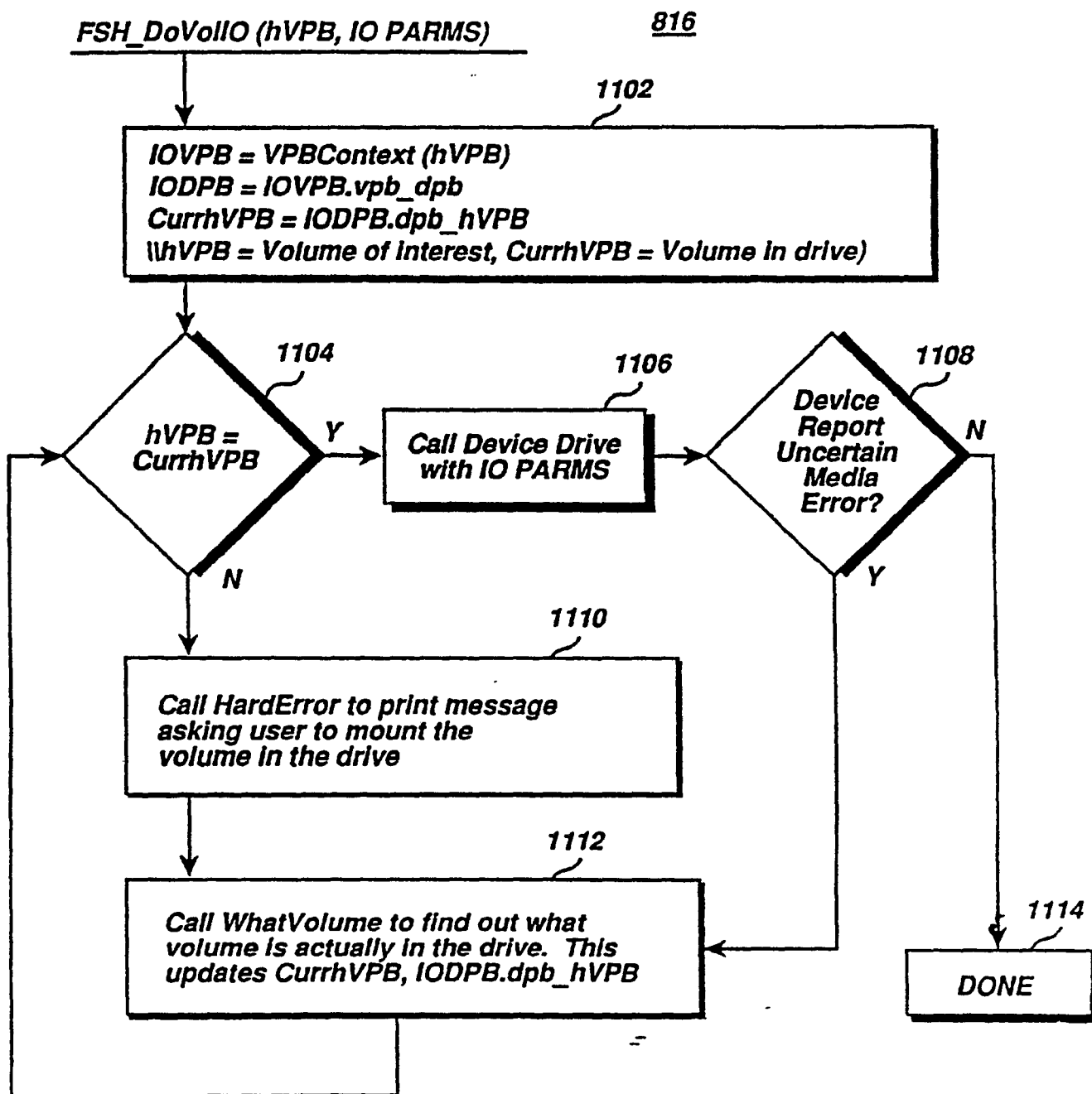


Figure 11